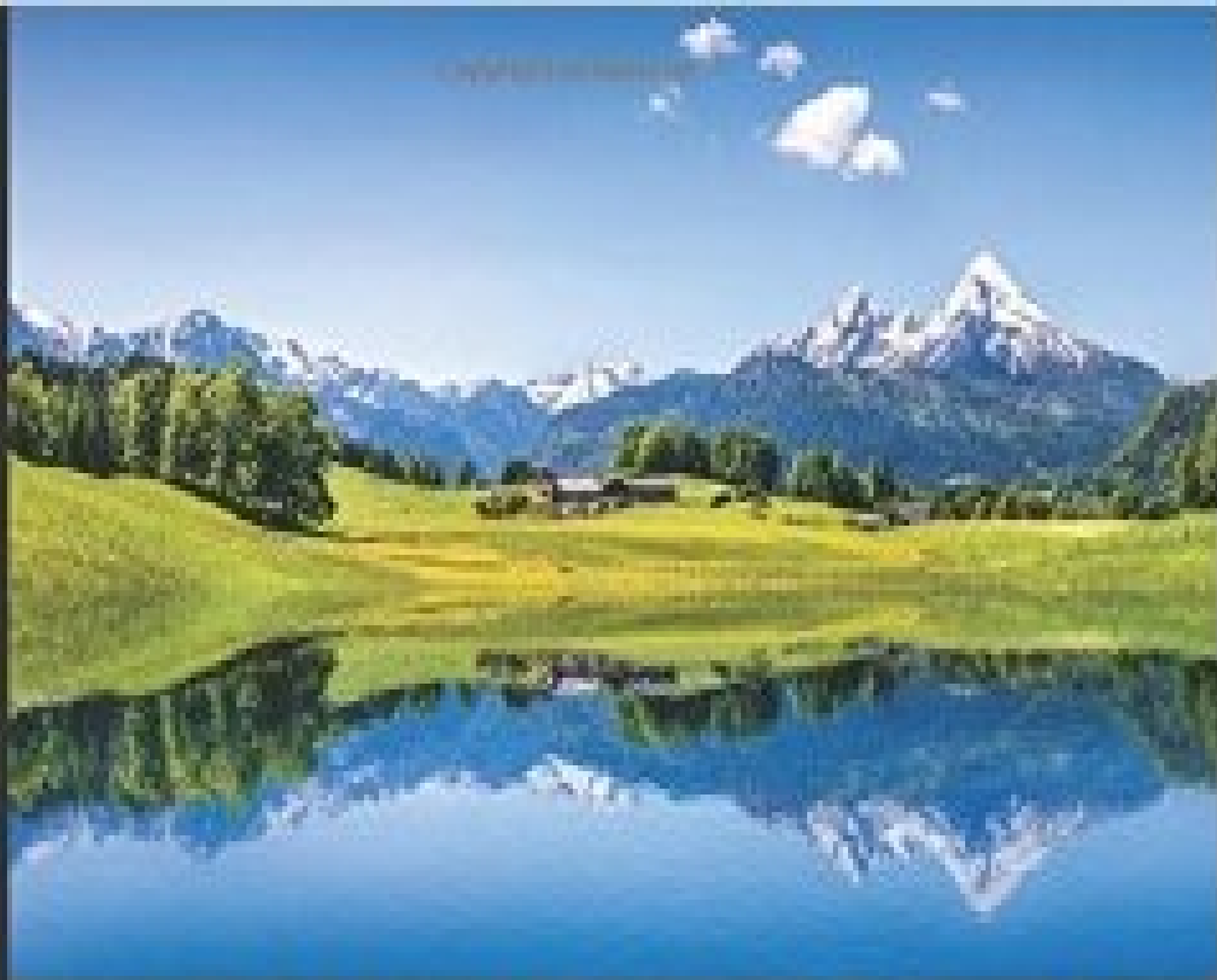


Community Experience Distilled

RxJava Essentials

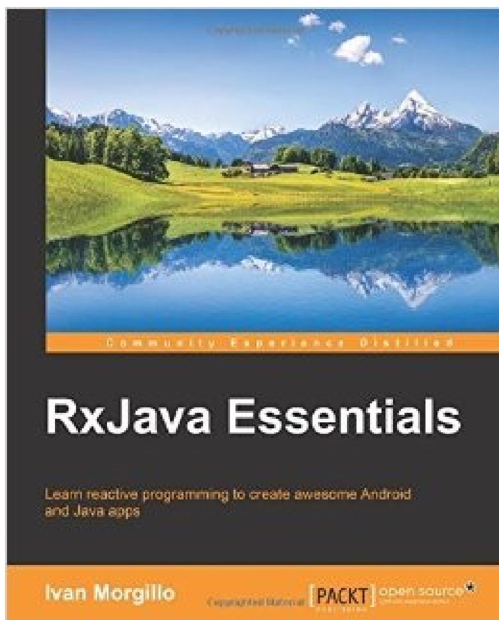
Learn reactive programming to create awesome Android and Java apps

Ivan Morgillo

[PACKT] open source
PUBLISHING

目錄

RxJava Essentials 中文翻译版



本书是对Ivan.Morgillo所写一书的中文翻译版本，仅供交流学习使用，严禁商业用途。另外推荐一本姊妹篇《Learning Reactive Programming》。

- 《RxJava Essentials》翻译中文版电子书
- 《RxJava Essentials》一书作者代码样例

本书内容有

1.RX-from .NET to RxJava

> 本章带你进入reactive的世界。我们会比较reactive 方法和传统方法，进而探索它们之间的相似和不同的地方。

2.Why Observables?

> 本章会对观察者模式做一个概述，如何实现它以及怎样用RxJava来进行扩展，被观察者是什么，以及被观察者如何与迭代联系到一起的。

3.Hello Reactive World

> 本章会利用我们所学的知识来创建第一个reactive Android应用。

4.Filtering Observables

> 本章我们会研究Observable序列的本质:filtering.我们也将学到如何从一个发出的Observable中选取我们想要的值，如何获得一个有限的数值，如何处理溢出的场景，以及更多有用的技巧。

5.Transforming Observables

> 本章将讲述如何通过变换Observable序列来创建出我们所需要的序列。

6.Combining Observables

> 本章将研究与函数结合，同时也会学到当创建我们想要的Observable时又如何与多个Observable协同工作。

7.Schedulers-Defeating the Android MainThread Issue

> 本章将介绍如何使用RxJava Schedulers 来处理多线程和并发编程。我们也将用reactive的方式来创建网络操作、内存访问、耗时处理。

8.REST in peace-RxJava and Retrofit

> 本章教会你如何让Square公司的Retrofit和RxJava结合来一起使用，来创建一个更高效的REST客户端程序。

学习这本书你需要做的：

为了能够运行书中的例子，你需要一个标准的Android开发环境：

- Android Studio 或 Intellij IDEA

- Android SDK
- Java SDK

作为一个纯粹的Java开发者，当你接触RxJava时，很明显你需要一个你喜欢Java编辑器和一个标准的Java JDK 环境。这本书中的一些图表来自 http://rxmarbles.com 和 http://reactivex.io 。

这本书适合哪些人看

如果你是一名有经验的Java开发者，reactive编程将会在后端系统中给你一种新的学习扩展和并发的方式，而这不需要更换开发语言。这本书将帮助你学习RxJava的核心方面,也能帮助你克服Android平台局限性从而创建一个基于事件驱动的，响应式的，流畅体验的Android应用。

一些约定

在这本书中，你会发现许多用来区分不同信息的文本样式，这列举这些样式的一些例子和对他们释义的说明。

以下列举了些文本中的代码、数据库表名、文件夹名、文件名、文件扩展名、路径名、伪造的URL、用户输入、Twitter handles ：“正如你看到的那样：zip()有三个参数：两个Observable和一个Func2”

如下面的一块代码：

```
public Observable<List<User>> getMostPopularSOusers(int howmany)
{
        return mStackExchangeService
                .getMostPopularSOusers(howmany)
                .map(UsersResponse::getUsers)
                .subscribeOn(Schedulers.io())
               .observeOn(AndroidSchedulers.mainThread());
}
```

当我们想对代码块的某一部分引起你的注意时，会在对应的那一行或列设置为粗体

```
public Observable<List<User>> getMostPopularSOusers(int howmany)
{
        return mStackExchangeService
                .getMostPopularSOusers(howmany)
                .map(UsersResponse::getUsers)   //也就是这句加粗显示

                .subscribeOn(Schedulers.io())
               .observeOn(AndroidSchedulers.mainThread());
}
```

新的项目和重要的词语都会以粗体显示。你在屏幕看到的字，例如在菜单或者对话框，会以类似这样的形式出现在文本中：“We will just need a fancy progress bar and a **DOWNLOAD** button.

Note

类似这样的是警告或者出现在框中的重要提示。

Tip 类似这样的是提示和技巧

读者反馈

发送邮件到 feedback@packtpub.com 在你的邮件主题中要提到书的标题。

如果你有擅长的话题并且你对写作感兴趣或者想出书的话，可以看我们作者指南：http://www.packtpub.com/authors

下载样例代码

你可以从你在http://www.packtpub.com的账户中下载所有你购买Packt 出版的图书的样例代码，如果你从别处购买这本书的话，你可以访问：http://www.packtpub.com/support 注册并将文件用附件直接发给你。

版权说明

RxJava Essentials 中文翻译版 仅供交流学习使用，严禁商业用途。转载请联系作者yuxingxin。

RX - 从.NET到RxJava

响应式编程是一种基于异步数据流概念的编程模式。数据流就像一条河：它可以被观测，被过滤，被操作，或者为新的消费者与另外一条流合并为一条新的流。

响应式编程的一个关键概念是事件。事件可以被等待，可以触发过程，也可以触发其它事件。事件是唯一的以合适的方式将我们的现实世界映射到我们的软件中：如果屋里太热了我们就打开一扇窗户。同样的，当我们更改电子表（变化的传播）中的一些数值时，我们需要更新整个表格或者我们的机器人碰到墙时会转弯（响应事件）。

今天，响应式编程最通用的一个场景是UI：我们的移动App必须做出对网络调用、用户触摸输入和系统弹框的响应。在这个世界上，软件之所以是事件驱动并响应的是因为现实生活也是如此。

微软响应式扩展

函数响应式编程是一个来自90年代后期受微软的一名计算机科学家Erik Meijer启发的思想，用来设计和开发微软的Rx库。

Rx 是微软.NET的一个响应式扩展。Rx借助可观测的序列提供一种简单的方式来创建异步的，基于事件驱动的程序。开发者可以使用Observables模拟异步数据流，使用LINQ语法查询Observables，并且很容易管理调度器的并发。

Rx让众所周知的概念变得易于实现和消费，例如**push**方法。在响应式的世界里，我们不能假装作用户不关注或者是不抱怨它而一味的等待函数的返回结果，网络调用，或者数据库查询的返回结果。我们时刻都在等待某些东西，这就让我们失去了并行处理其他事情的机会，提供更好的用户体验，让我们的软件免受顺序链的影响，而阻塞编程。

下表列出的与.NET 枚举相关的.NET Observable

.NET Observable	一个返回值	多个返回值
Pull/Synchronous/Interactive	`T`	`IEnumerable<T>`
Push/Asynchronous/Reactive	`Task<T>`	`IObservable<T>`

push方法把这个问题逆转了：取而代之的是不再等待结果，开发者只是简单的请求结果，而当它返回时得到一个通知即可。开发者对即将发生的事件提供一个清晰的响应链。对于每一个事件，开发者都作出相应的响应；例如，用户被要求登录的时候，提交一个携带他的用户名和密码的表单。应用程序执行登录的网络请求，接下来将要发生的情况有：

- 显示一个成功的信息，并保存用户的个人信息。
- 显示一个错误的信息

正如你用push方法所看到的，开发者不需要等待结果。而是在结果返回时通知他。在这期间，他可以做他想做的任何事情：

- 显示一个进度对话框
- 为下次登录保存用户名和密码
- 预加载一些他认为登录成功后需要耗时处理的事情

来到Java世界 - Netflix RxJava

Netflix在2012年开始意识到他们的架构要满足他们庞大的用户群体已经变得步履维艰。因此他们决定重新设计架构来减少REST调用的次数。取代几十次的REST调用，而是让客户端自己处理需要的数据，他们决定基于客户端需求创建一个专门优化过的REST调用。

为了实现这一目标，他们决定尝试响应式，开始将.NET Rx迁移到JVM上面。他们不想只基于Java语言；而是整个JVM，从而有可能为市场上的每一种基于JVM的语言：如Java、Clojure、Groovy、Scala等等提供一种新的工具。

2013年二月份,Ben Christensen 和 Jafar Husain发在Netflix技术博客的一篇文章第一次向世界展示了RxJava。

主要特点有：

- 易于并发从而更好的利用服务器的能力。
- 易于有条件的异步执行。
- 一种更好的方式来避免回调地狱。
- 一种响应式方法。

正如.NET,RxJava Observable 是push 迭代的等价体，即pull。pull方法是阻塞并等待的方法：消费者从源头pull值，并阻塞线程直到生产者提供新的值。

push方法作用于订阅和响应：消费者订阅新值的发射，当它们可用时生产者push这些新值并通知消费者。在这一点上，消费者消费了它们。push方法很明显更灵活，因为从逻辑和实践的观点来看，开发者只需忽略他需要的数据是来自同步还是异步；他的代码将仍然起作用。

RxJava的与众不同之处

从纯Java的观点看，RxJava Observable类源自于经典的Gang Of Four的观察者模式。

它添加了三个缺少的功能：

- 生产者在没有更多数据可用时能够发出信号通知：onCompleted()事件。
- 生产者在发生错误时能够发出信号通知：onError()事件。
- RxJava Observables 能够组合而不是嵌套，从而避免开发者陷入回调地狱。

Observables和Iterables共用一个相似的API：我们在Iterable可以执行的许多操作也都同样可以在Observables上执行。当然，由于Observables流的本质，没有如Iterable.remove()这样相应的方法。

Pattern	一个返回值	多个返回值
Synchronous	`T getData()`	`Iterable<T>`
Asynchronous	`Future<T> getData()`	`Observable<T> getData()`

从语义的角度来看，RxJava就是.NET Rx。从语法的角度来看，Netflix考虑到了对应每个Rx方法,保留了Java代码规范和基本的模式。

总结

本章中，我们初步探索了响应式的世界。从微软的.NET到Netflix的RxJava，我们了解了Rx是如何诞生的，我们也了解到传统的方法与响应式方法相比之间的相似和不同。

下一章，我们将学习到Observables是什么，以及如何创建它并把响应式编程应用到我们的日常编码中去。

为什么是Observables?

在面向对象的架构中，开发者致力于创建一组解耦的实体。这样的话，实体就可以在不用妨碍整个系统的情况下可以被测试、复用和维护。设计这种系统就带来一个棘手的负面影响：维护相关对象之间的统一。

在Smalltalk MVC架构中，创建模式的第一个例子就是用来解决这个问题的。用户界面框架提供一种途径使UI元素与包含数据的实体对象相分离，并且同时，它提供一种灵活的方法来保持它们之间的同步。

在这本畅销的四人组编写的《设计模式——可复用面向对象软件的基础》一书中，观察者模式是最有名的设计模式之一。它是一种行为模式并提供一种以一对多的依赖来绑定对象的方法：即当一个对象发生变化时，依赖它的所有对象都会被通知并且会自动更新。

在本章中，我们将会对观察者模式有一个概述，它是如何实现的以及如何用RxJava来扩展，Observable是什么，以及Observables如何与Iterables相关联。

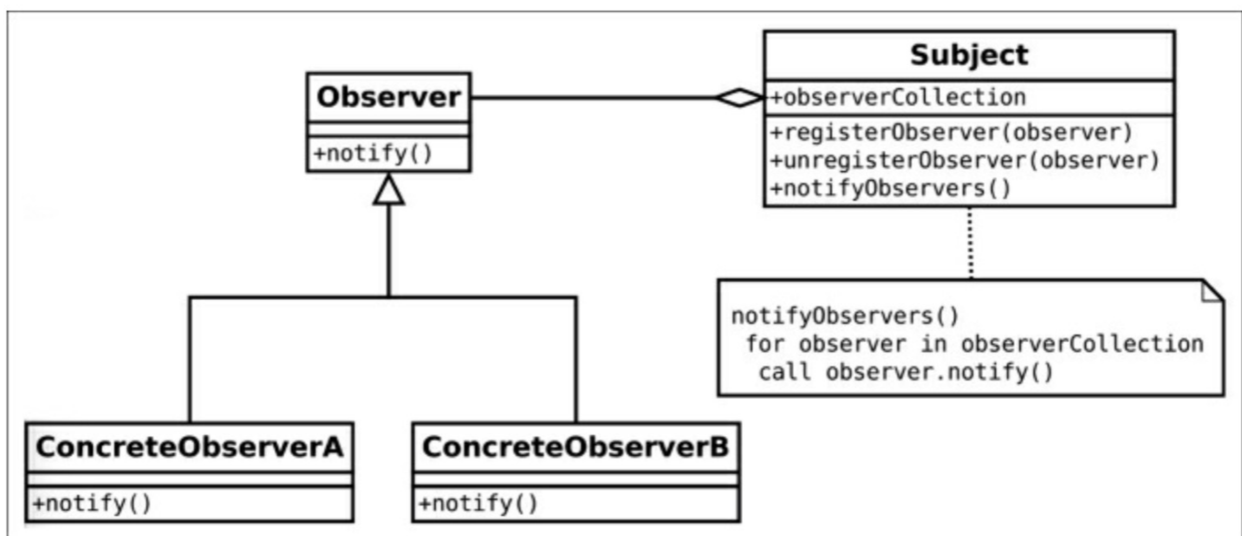
观察者模式

在今天，观察者模式是出现的最常用的软件设计模式之一。它基于subject这个概念。subject是一种特殊对象，当它改变时，那些由它保存的一系列对象将会得到通知。而这一系列对象被称作Observers,它们会对外暴漏了一个通知方法,当subject状态发生变化时会调用的这个方法。

在上一章中，我们看到了电子表单的例子。现在我们可以展开这个例子讲，展示一个更复杂的场景。让我们考虑这样一个填着账户数据的电子表单。我们可以把这些数据比作一张表，或者是3D柱状图，或者是饼状图。它们中每一个代表的意义都取决于同一组要展示的数据。每一个都是一个观察者，都依赖于那一个subject，维护着全部信息。

3D柱状图这个类、饼状图类、表这个类以及维护这些数据的类是完全解耦的：它们彼此相互独立复用，但也能协同工作。这些表示类彼此不清楚对方，但是正如它们所做的：它们知道在哪能找到它们需要展示的信息，它们也知道一旦数据发生变化就通知需要更新数据表示的那个类。

这有一张图描述了Subject/Observer的关系是怎样的一对多的关系：



上面这张图展示了一个Subject为3个Observers提供服务。很明显，没有理由去限制Observers的数量：如果有需要，一个Subject可以有无限多个Observers,当subject状态发生变化时，这些Observers中的每一个都会收到通知。

你什么时候使用观察者模式？

观察者模式很适合下面这些场景中的任何一个：

- 当你的架构有两个实体类，一个依赖另一个，你想让它们互不影响或者是独立复用它们时。
- 当一个变化的对象通知那些与它自身变化相关联的未知数量的对象时。
- 当一个变化的对象通知那些无需推断具体是谁的对象时。

RxJava观察者模式工具包

在RxJava的世界里，我们有四种角色：

- Observable
- Observer
- Subscriber
- Subjects

Observables和Subjects是两个“生产”实体，Observers和Subscribers是两个“消费”实体。

Observable

当我们异步执行一些复杂的事情，Java提供了传统的类，例如Thread、Future、FutureTask、CompletableFuture来处理这些问题。当复杂度提升，这些方案就会变得麻烦和难以维护。最糟糕的是，它们都不支持链式调用。

RxJava Observables被设计用来解决这些问题。它们灵活，且易于使用，也可以链式调用，并且可以作用于单个结果程序上，更有甚者，也可以作用于序列上。无论何时你想发射单个标量值，或者一连串值，甚至是无穷个数值流，你都可以使用Observable。

Observable的生命周期包含了三种可能的易于与Iterable生命周期事件相比较的事件，下表展示了如何将Observable async/push 与 Iterable sync/pull相关联起来。

Event	Iterable(pull)	Observable(push)
检索数据	`T next()`	`onNext(T)`
发现错误	`throws Exception`	`onError(Throwable)`
完成	`!hasNext()`	`onCompleted()`

使用Iterable时，消费者从生产者那里以同步的方式得到值，在这些值得到之前线程处于阻塞状态。相反，使用Observable时，生产者以异步的方式把值推给观察者，无论何时，这些值都是可用的。这种方法之所以更灵活是因为即便值是同步或异步方式到达，消费者在这两种场景都可以根据自己的需要来处理。

为了更好地复用Iterable接口，RxJava Observable类扩展了GOF观察者模式的语义。引入了两个新的接口：

- onCompleted() 即通知观察者Observable没有更多的数据。
- onError() 即观察者有错误出现了。

热Observables和冷Observables

从发射物的角度来看，有两种不同的Observables:热的和冷的。一个"热"的Observable典型的只要一创建完就开始发射数据，因此所有后续订阅它的观察者可能从序列中间的某个位置开始接受数据（有一些数据错过了）。一个"冷"的

Observable会一直等待，直到有观察者订阅它才开始发射数据，因此这个观察者可以确保会收到整个数据序列。

创建一个Observable

在接下来的小节中将讨论Observables提供的两种创建Observable的方法。

Observable.create()

create()方法使开发者有能力从头开始创建一个Observable。它需要一个OnSubscribe对象,这个对象继承Action1,当观察者订阅我们的Observable时，它作为一个参数传入并执行call()函数。

```
Observable.create(new Observable.OnSubscribe<Object>(){
        @Override
        public void call(Subscriber<? super Object> subscriber)
{

        }
});
```

Observable通过使用subscriber变量并根据条件调用它的方法来和观察者通信。让我们看一个“现实世界”的例子：

```
Observable<Integer> observableString = Observable.create(new Obs
ervable.OnSubscribe<Integer>() {
        @Override
        public void call(Subscriber<? super Integer> observer) {
            for (int i = 0; i < 5; i++) {
                observer.onNext(i);
            }
            observer.onCompleted();
        }
});

Subscription subscriptionPrint = observableString.subscribe(new
Observer<Integer>() {
    @Override
    public void onCompleted() {
        System.out.println("Observable completed");
    }

    @Override
    public void onError(Throwable e) {
        System.out.println("Oh,no! Something wrong happened！");
    }

    @Override
    public void onNext(Integer item) {
        System.out.println("Item is " + item);
    }
});
```

例子故意写的简单，是因为即便是你第一次见到RxJava的操作，我想让你明白接下来要发生什么。

我们创建一个新的 `Observable<Integer>` ,它执行了5个元素的for循环，一个接一个的发射他们，最后完成。

另一方面，我们订阅了Observable，返回一个Subscription 。一旦我们订阅了，我们就开始接受整数，并一个接一个的打印出它们。我们并不知道要接受多少整数。事实上，我们也无需知道是因为我们为每种场景都提供对应的处理操作：

- 如果我们接收到了整数，那么就打印它。
- 如果序列结束，我们就打印一个关闭的序列信息。
- 如果错误发生了，我们就打印一个错误信息。

Observable.from()

在上一个例子中，我们创建了一个整数序列并一个一个的发射它们。假如我们已经有一个列表呢？我们是不是可以不用for循环而也可以一个接一个的发射它们呢？

在下面的例子代码中，我们从一个已有的列表中创建一个Observable序列：

```
List<Integer> items = new ArrayList<Integer>();
items.add(1);
items.add(10);
items.add(100);
items.add(200);

Observable<Integer> observableString = Observable.from(items);
Subscription subscriptionPrint = observableString.subscribe(new
Observer<Integer>() {
    @Override
    public void onCompleted() {
        System.out.println("Observable completed");
    }

    @Override
    public void onError(Throwable e) {
        System.out.println("Oh,no! Something wrong happened！");
    }

    @Override
    public void onNext(Integer item) {
        System.out.println("Item is " + item);
    }
});
```

输出的结果和上面的例子绝对是一样的。

`from()` 创建符可以从一个列表/数组来创建Observable,并一个接一个的从列表/数组中发射出来每一个对象，或者也可以从Java `Future` 类来创建Observable，并发射Future对象的 `.get()` 方法返回的结果值。传入 `Future` 作为参数时，我们可以指定一个超时的值。Observable将等待来自 `Future` 的结果；如果在超时之前仍然没有结果返回，Observable将会触发 `onError()` 方法通知观察者有错误发生了。

Observable.just()

如果我们已经有了一个传统的Java函数，我们想把它转变为一个Observable又改怎么办呢？我们可以用 `create()` 方法，正如我们先前看到的，或者我们也可以像下面那样使用以此来省去许多模板代码：

```
Observable<String> observableString = Observable.just(helloWorld
());

Subscription subscriptionPrint = observableString.subscribe(new
Observer<String>() {
    @Override
    public void onCompleted() {
        System.out.println("Observable completed");
    }

    @Override
    public void onError(Throwable e) {
        System.out.println("Oh,no! Something wrong happened!");
    }

    @Override
    public void onNext(String message) {
        System.out.println(message);
    }
});
```

`helloWorld()` 方法比较简单，像这样：

```
private String helloWorld(){
    return "Hello World";
}
```

不管怎样，它可以是我们想要的任何函数。在刚才的例子中，我们一旦创建了Observable， `just()` 执行函数，当我们订阅Observable时，它就会发射出返回的值。

`just()` 方法可以传入一到九个参数，它们会按照传入的参数的顺序来发射它们。 `just()` 方法也可以接受列表或数组，就像 `from()` 方法，但是它不会迭代列表发射每个值,它将会发射整个列表。通常，当我们想发射一组已经定义好的值时会用到它。但是如果我们的函数不是时变性的，我们可以用just来创建一个更有组织性和可测性的代码库。

最后注意 `just()` 创建符，它发射出值后，Observable正常结束，在上面那个例子中，我们会在控制台打印出两条信息：“Hello World”和“Observable completed”。

Observable.empty(),Observable.never(),和Observable.throw()

当我们需要一个Observable毫无理由的不再发射数据正常结束时，我们可以使用 `empty()` 。我们可以使用 `never()` 创建一个不发射数据并且也永远不会结束的Observable。我们也可以使用 `throw()` 创建一个不发射数据并且以错误结束的Observable。

Subject = Observable + Observer

`subject` 是一个神奇的对象，它可以是一个Observable同时也可以是一个Observer：它作为连接这两个世界的一座桥梁。一个Subject可以订阅一个Observable，就像一个观察者，并且它可以发射新的数据，或者传递它接受到的数据，就像一个Observable。很明显，作为一个Observable，观察者们或者其它Subject都可以订阅它。

一旦Subject订阅了Observable，它将会触发Observable开始发射。如果原始的Observable是“冷”的，这将会对订阅一个“热”的Observable变量产生影响。

RxJava提供四种不同的Subject：

- PublishSubject
- BehaviorSubject
- ReplaySubject.
- AsyncSubject

PublishSubject

Publish是Subject的一个基础子类。让我们看看用PublishSubject实现传统的Observable `Hello World` :

```
PublishSubject<String> stringPublishSubject = PublishSubject.cre
ate();
Subscription subscriptionPrint = stringPublishSubject.subscribe(
new Observer<String>() {
    @Override
    public void onCompleted() {
        System.out.println("Observable completed");
    }

    @Override
    public void onError(Throwable e) {
        System.out.println("Oh,no!Something wrong happened!");

    }

    @Override
    public void onNext(String message) {
        System.out.println(message);
    }
});
stringPublishSubject.onNext("Hello World");
```

在刚才的例子中，我们创建了一个 `PublishSubject` ，用 `create()` 方法发射一个 `String` 值，然后我们订阅了PublishSubject。此时，没有数据要发送，因此我们的观察者只能等待，没有阻塞线程，也没有消耗资源。就在这随时准备从subject接收值，如果subject没有发射值那么我们的观察者就会一直在等待。再次声明的是，无需担心：观察者知道在每个场景中该做什么，我们不用担心什么时候是因为它是响应式的：系统会响应。我们并不关心它什么时候响应。我们只关心它响应时该做什么。

最后一行代码展示了手动发射字符串“Hello World”,它触发了观察者的 `onNext()` 方法，让我们在控制台打印出“Hello World”信息。

让我们看一个更复杂的例子。话说我们有一个 `private` 声明的Observable，外部不能访问。Observable在它生命周期内发射值，我们不用关心这些值，我们只关心他们的结束。

首先，我们创建一个新的PublishSubject来响应它的 `onNext()` 方法，并且外部也可以访问它。

```
final PublishSubject<Boolean> subject = PublishSubject.create();

subject.subscribe(new Observer<Boolean>() {
    @Override
    public void onCompleted() {

    }

    @Override
    public void onError(Throwable e) {

    }

    @Override
    public void onNext(Boolean aBoolean) {
        System.out.println("Observable Completed");
    }
});
```

然后，我们创建“私有”的Observable，只有subject才可以访问的到。

```
Observable.create(new Observable.OnSubscribe<Integer>() {
    @Override
    public void call(Subscriber<? super Integer> subscriber) {
        for (int i = 0; i < 5; i++) {
            subscriber.onNext(i);
        }
        subscriber.onCompleted();
    }
}).doOnCompleted(new Action0() {
    @Override
    public void call() {
        subject.onNext(true);
    }
}).subscribe();
```

`Observable.create()` 方法包含了我们熟悉的for循环，发射数字。 `doOnCompleted()` 方法指定当Observable结束时要做什么事情：在subject上发射true。最后，我们订阅了Observable。很明显，空的 `subscribe()` 调用仅仅是为了开启Observable，而不用管已发出的任何值，也不用管完成事件或者错误事件。为了这个例子我们需要它像这样。

在这个例子中，我们创建了一个可以连接Observables并且同时可被观测的实体。当我们想为公共资源创建独立、抽象或更易观测的点时，这是极其有用的。

BehaviorSubject

简单的说，BehaviorSubject会首先向他的订阅者发送截至订阅前最新的一个数据对象（或初始值）,然后正常发送订阅后的数据流。

```
BehaviorSubject<Integer> behaviorSubject = BehaviorSubject.create(1);
```

在这个短例子中，我们创建了一个能发射整形(Integer)的BehaviorSubject。由于每当Observes订阅它时就会发射最新的数据，所以它需要一个初始值。

ReplaySubject

ReplaySubject会缓存它所订阅的所有数据,向任意一个订阅它的观察者重发:

```
ReplaySubject<Integer> replaySubject = ReplaySubject.create();
```

AsyncSubject

当Observable完成时AsyncSubject只会发布最后一个数据给已经订阅的每一个观察者。

```
AsyncSubject<Integer> asyncSubject = AsyncSubject.create();
```

总结

本章中，我们了解到了什么是观察者模式，为什么Observables在今天的编程场景中如此重要，以及如何创建Observables和subjects。

下一章中，我们将创建第一个基于RxJava的Android应用程序，学习如何检索数据来填充listview，以及探索如何创建一个基于RxJava的响应式UI。

向响应式世界问好

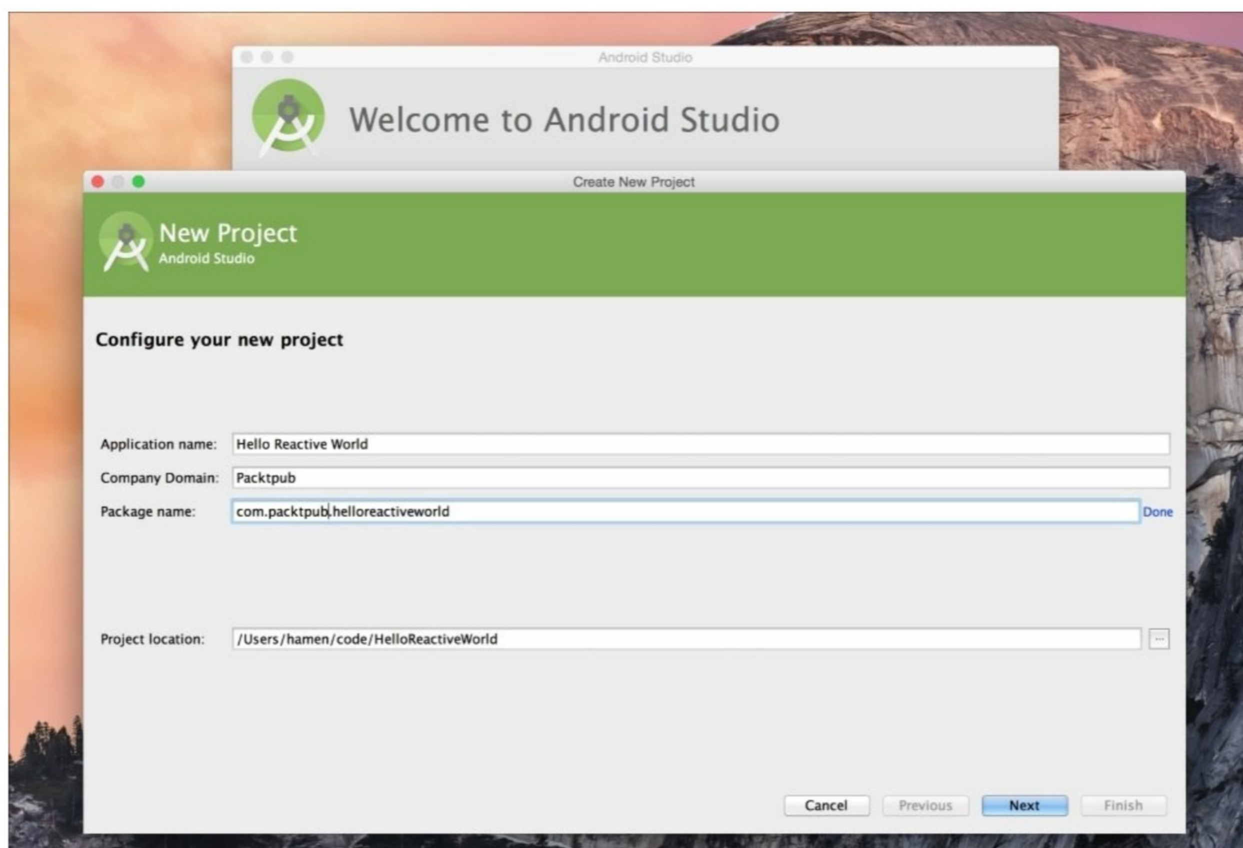
在上一章中，我们对观察者模式有个理论上的快速概述。我们也看了从头开始、从列表、或者从已经存在的函数来创建Observables。在本章中，我们将用我们学到的来创建我们第一个响应式Android应用程序。首先，我们需要搭建环境，导入需要的库和有用的库。然后我们将创建一个简单的应用程序，在不同的flavors中包含几个用RxJava填充的RecycleView items。

启动引擎

我们将使用IntelliJ IDEA/Android Studio来创建这个工程，因此你会对截图看起来比较熟悉。

让我们开始创建一个新的Android工程。你可以创建你自己的工程或者用本书中提供的导入。选择你自己喜欢的创建方式这取决于你。

如果你想用Android Studio创建一个新的工程，通常你可以参考官方文档：http://developer.android.com/intl/zh-cn/training/basics/firstapp/creating-project.html



依赖

很明显，我们将使用**Gradle**来管理我们的依赖列表。我们的build.gradle文件看起来像这样：

```
MainActivity.java × | FirstExampleFragment.java × | app × | fragment_first_example.xml ×

buildscript {
    repositories {
        mavenCentral()
    }

    dependencies {
        classpath 'me.tatarka:gradle-retrolambda:2.5.0'
    }
}

repositories {
    mavenCentral()
}

apply plugin: 'com.android.application'
apply plugin: 'me.tatarka.retrolambda'

android {
    compileSdkVersion 21
    buildToolsVersion "21.1.2"

    defaultConfig {
        applicationId "com.packtpub.apps.rxjava_essentials"
        minSdkVersion 16
        targetSdkVersion 21
        versionCode 1
        versionName "1.0"
    }

    buildTypes {
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rules.pro'
        }
    }

    compileOptions {
        sourceCompatibility JavaVersion.VERSION_1_8
        targetCompatibility JavaVersion.VERSION_1_8
    }

    lintOptions {
        disable 'InvalidPackage'
    }

    packagingOptions {
        exclude 'META-INF/services/javax.annotation.processing.Processor'
    }
}

dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    compile 'com.android.support:support-v4:21.0.3'
    compile "com.android.support:appcompat-v7:21.0.3"
    compile 'com.android.support:recyclerview-v7:21.0.0'
    compile 'com.android.support:cardview-v7:21.0.3'

    compile 'org.projectlombok:lombok:1.14.8'
    compile 'com.jakewharton:butterknife:6.0.0'

    compile 'io.reactivex:rxandroid:0.24.0'
}
```

正如你看到的我们引入了RxAndroid。RxAndroid是RxJava的增强版，尤其是针对Android设计的。

RxAndroid

RxAndroid是RxJava家族的一部分。它基于RxJava1.0.x,在普通的RxJava基础上添加了几个有用的类。大多数情况下，它为Android添加了特殊的调度器。我们将在第七章Schedulers-Defeating the Android MainThread Issue再讨论它。

工具

出于实用，我们引入了Lombok 和 Butter Knife。这两个可以帮助我们在Android应用程序中少写许多模板类代码。

Lombok

Lombok使用注解的方式为你生成许多代码。大多数情况下我们使用其生成 `getter/setter` 、 `toString()` 、 `equals()` 、 `hashCode()` 。它来自于一个Gradle依赖和Android Studio插件。

Butter Knife

Butter Knife使用注解的方式来帮助我们免去写 `findViewById()` 和设置点击监听的痛苦。至于Lombok,我们可以通过导入依赖和安装Android Studio插件来获得更好的体验。

Retrolambda

最后，我们导入Retrolambda，是因为我们开发的Android是基于Java 1.6，然后我们可以借助它来实现Java 8 Lambda函数从而减少许多模板代码。

我们的第一个Observable

在我们的第一个列子里，我们将检索安装的应用列表并填充RecycleView的item来展示它们。我们也设想一个下拉刷新的功能和一个进度条来告知用户当前任务正在执行。

首先，我们创建Observable。我们需要一个函数来检索安装的应用程序列表并把它提供给我们的观察者。我们一个接一个的发射这些应用程序数据，将它们分组到一个单独的列表中，以此来展示响应式方法的灵活性。

```
private Observable<AppInfo> getApps(){
    return Observable.create(subscriber -> {
        List<AppInfoRich> apps = new ArrayList<AppInfoRich>();

        final Intent mainIntent = new Intent(Intent.ACTION_MAIN,
null);
        mainIntent.addCategory(Intent.CATEGORY_LAUNCHER);

        List<ResolveInfo> infos = getActivity().getPackageManage
r().queryIntentActivities(mainIntent, 0);

        for(ResolveInfo info : infos){
            apps.add(new AppInfoRich(getActivity(),info));
        }

        for (AppInfoRich appInfo:apps) {
            Bitmap icon = Utils.drawableToBitmap(appInfo.getIcon
());
            String name = appInfo.getName();
            String iconPath = mFilesDir + "/" + name;
            Utils.storeBitmap(App.instance, icon,name);

            if (subscriber.isUnsubscribed()){
                return;
            }
            subscriber.onNext(new AppInfo(name,iconPath,appInfo.
getLastUpdateTime()));
        }
        if (!subscriber.isUnsubscribed()){
            subscriber.onCompleted();
        }
    });
}
```

AppInfo对象如下：

```
@Data
@Accessors(prefix = "m")
public class AppInfo implements Comparable<Object> {

    long mLastUpdateTime;
    String mName;
    String mIcon;

    public AppInfo(String nName, long lastUpdateTime, String ico
n) {
        mName = nName;
        mIcon = icon;
        mLastUpdateTime = lastUpdateTime;
    }

    @Override
    public int compareTo(Object another) {
        AppInfo f = (AppInfo)another;
        return getName().compareTo(f.getName());
    }
}
```

需要重点注意的是在发射新的数据或者完成序列之前要检测观察者的订阅情况。这样的话代码会更高效，因为如果没有观察者等待时我们就不生成没有必要的数据项。

此时，我们可以订阅Observable并观察它。订阅一个Observable意味着当我们需要的数据进来时我们必须提供对应的操作来执行它。

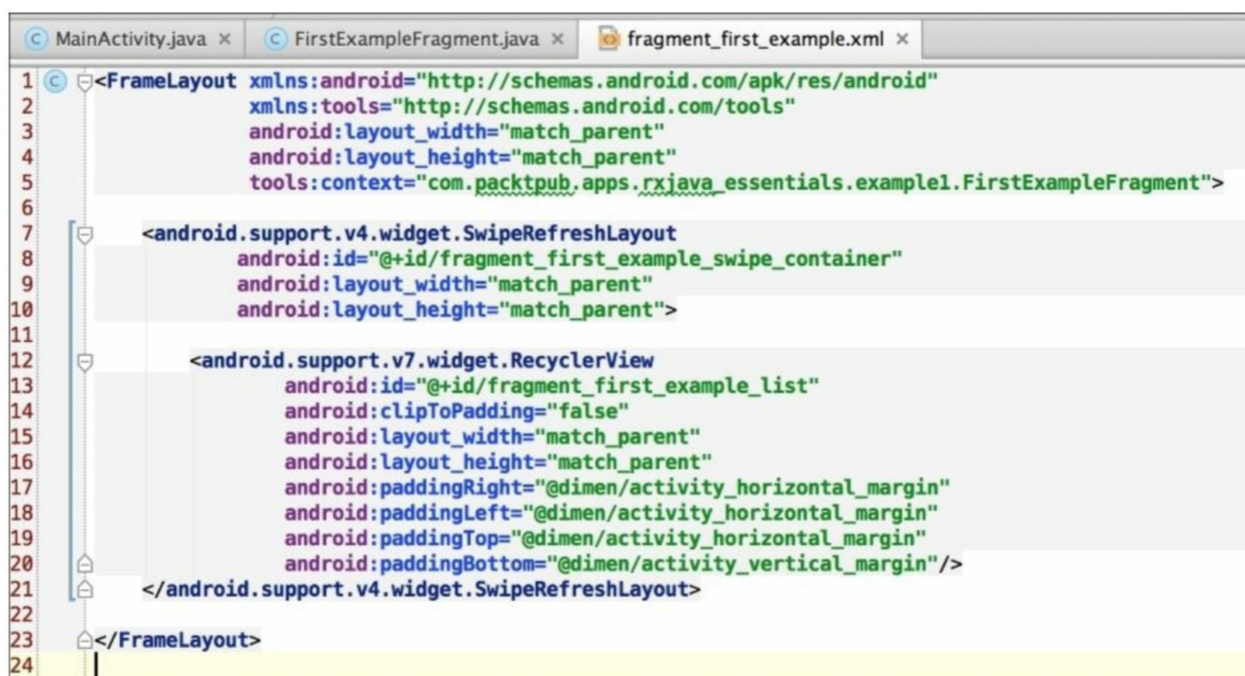
当前的场景是什么？我们展示一个进度条来等待数据。当数据到来时，我们需要隐藏掉进度条,填充list,最终展示列表。现在，我们知道当一切都准备好了该做什么。那么错误的场景呢？对于错误这种情况，我们仅仅是用Toast展示一个错误的信息。

使用Butter Knife,我们得到list和下拉刷新组件的引用：

```
@InjetcView(R.id.fragment_first_example_list)
RecyclerView mRecycleView;

@InjectView(R.id.fragment_first_example_swipe_container)
SwipeRefreshLayout mSwipeRefreshLayout;
```

我们使用Android 5的标准组件：RecyclerView和SwipeRefreshLayout。截屏展示了我们这个简单App的list Fragment的layout文件：



```
MainActivity.java    FirstExampleFragment.java    fragment_first_example.xml

<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
             xmlns:tools="http://schemas.android.com/tools"
             android:layout_width="match_parent"
             android:layout_height="match_parent"
             tools:context="com.packtpub.apps.rxjava_essentials.example1.FirstExampleFragment">

    <android.support.v4.widget.SwipeRefreshLayout
            android:id="@+id/fragment_first_example_swipe_container"
            android:layout_width="match_parent"
            android:layout_height="match_parent">

        <android.support.v7.widget.RecyclerView
                android:id="@+id/fragment_first_example_list"
                android:clipToPadding="false"
                android:layout_width="match_parent"
                android:layout_height="match_parent"
                android:paddingRight="@dimen/activity_horizontal_margin"
                android:paddingLeft="@dimen/activity_horizontal_margin"
                android:paddingTop="@dimen/activity_horizontal_margin"
                android:paddingBottom="@dimen/activity_vertical_margin"/>
    </android.support.v4.widget.SwipeRefreshLayout>

</FrameLayout>
```

我们使用一个下拉刷新方法，因此列表数据可以来自初始化加载，或由用户触发的一个刷新动作。针对这两个场景，我们用同样的行为，因此我们把我们的观察者放在一个易被复用的函数里面。下面是我们的观察者，定义了成功、失败、完成要做的事情：

```
private void refreshTheList() {
    getApps().toSortedList()
            .subscribe(new Observer<List<AppInfo>>() {

                @Override
                public void onCompleted() {
                    Toast.makeText(getActivity(), "Here is the l
ist!", Toast.LENGTH_LONG).show();
                }

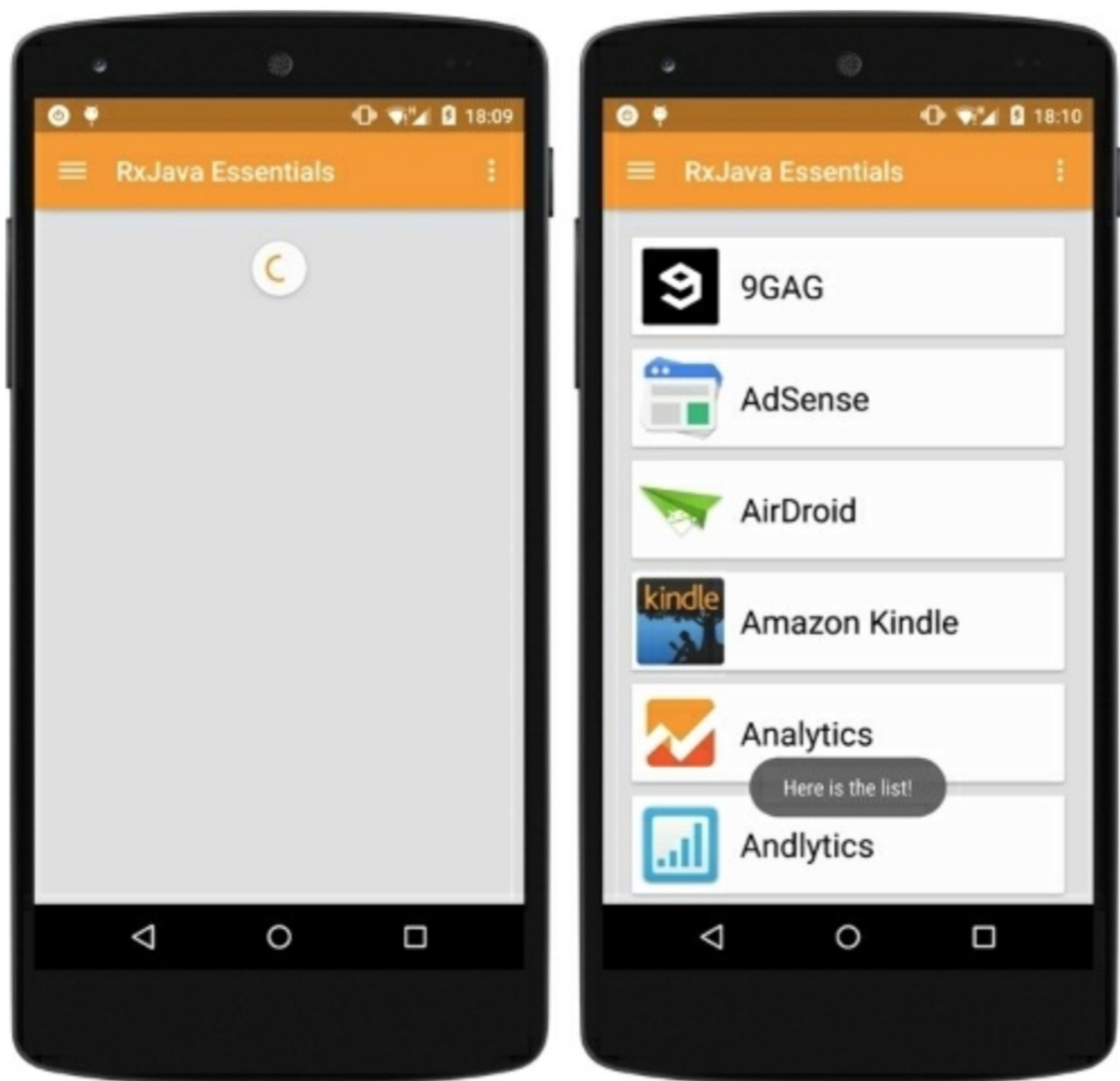
                @Override
                public void onError(Throwable e) {
                    Toast.makeText(getActivity(), "Something wen
t wrong!", Toast.LENGTH_SHORT).show();
                    mSwipeRefreshLayout.setRefreshing(false);
                }

                @Override
                public void onNext(List<AppInfo> appInfos) {
                    mRecyclerView.setVisibility(View.VISIBLE);
                    mAdapter.addApplications(appInfos);
                    mSwipeRefreshLayout.setRefreshing(false);
                }
            });
}
```

定义一个函数使我们能够用同样一个block来处理两种场景成为了可能。当fragment加载时我们只需调用 `refreshTheList()` 方法并设置 `refreshTheList()` 方法作为用户下拉这一行为所触发的方法。

```
mSwipeRefreshLayout.setOnRefreshListener(this::refreshTheList);
```

我们第一个例子现在完成了，运行跑一下。


18:09
RxJava Essentials
18:10
RxJava Essentials
9GAG
AdSense
AirDroid
kindle
Amazon Kindle
Analytics
Here is the list!
Andlytics

从列表创建一个Observable

在这个例子中，我们将引入 `from()` 函数。使用这个特殊的“创建”函数，我们可以从一个列表中创建一个Observable。Observable将发射出列表中的每一个元素，我们可以通过订阅它们来对这些发出的元素做出响应。

为了实现和第一个例子同样的结果，我们在每一个 `onNext()` 函数更新我们的适配器，添加元素并通知插入。

我们将复用和第一个例子同样的结构。主要的不同的是我们不再检索已安装的应用列表。列表由外部实体提供：

```
mApps = ApplicationsList.getInstance().getList();
```

获得列表后，我们仅需将它响应化并填充RecyclerView的item:

```
private void loadList(List<AppInfo> apps) {
    mRecyclerView.setVisibility(View.VISIBLE);
    Observable.from(apps)
            .subscribe(new Observer<AppInfo>() {

                @Override
                public void onCompleted() {
                    mSwipeRefreshLayout.setRefreshing(false);
                    Toast.makeText(getActivity(), "Here is the l
ist!", Toast.LENGTH_LONG).show();
                }

                @Override
                public void onError(Throwable e) {
                    Toast.makeText(getActivity(), "Something wen
t wrong!", Toast.LENGTH_SHORT).show();
                    mSwipeRefreshLayout.setRefreshing(false);
                }

                @Override
                public void onNext(AppInfo appInfo) {
                    mAddedApps.add(appInfo);
                    mAdapter.addApplication(mAddedApps.size() - 1
,appInfo);
                }
            });
}
```

正如你看到的，我们将已安装的应用程序列表作为参数传进 `from()` 函数，然后我们订阅生成的Observable。观察者和我们第一个例子中的观察者十分相像。一个主要的不同是我们在 `onCompleted()` 函数中停掉进度条是因为我们一个一个的发射元素；第一个例子中的Observable发射的是整个list,因此在 `onNext()` 函数中停掉进度条的做法是安全的。

再多几个例子

在这一节中，我们将基于RxJava的 `just()` , `repeat()` , `defer()` , `range()` , `interval()` ,和 `timer()` 方法展示一些例子。

just()

假如我们只有3个独立的AppInfo对象并且我们想把他们转化为Observable并填充到RecyclerView的item中：

```
List<AppInfo> apps = ApplicationsList.getInstance().getList();

AppInfo appOne = apps.get(0);

AppInfo appTwo = apps.get(10);

AppInfo appThree = apps.get(24);

loadApps(appOne,appTwo,appThree);
```

我们可以像我们之前的例子那样检索列表并提取出这三个元素。然后我们将他们传到这个 `loadApps()` 函数里面：

```
private void loadApps(AppInfo appOne,AppInfo appTwo,AppInfo appT
hree) {
    mRecyclerView.setVisibility(View.VISIBLE);
    Observable.just(appOne,appTwo,appThree)
            .subscribe(new Observer<AppInfo>() {

                @Override
                public void onCompleted() {
                    mSwipeRefreshLayout.setRefreshing(false);
                    Toast.makeText(getActivity(), "Here is the l
ist!", Toast.LENGTH_LONG).show();
                }

                @Override
                public void onError(Throwable e) {
                    Toast.makeText(getActivity(), "Something wen
t wrong!", Toast.LENGTH_SHORT).show();
                    mSwipeRefreshLayout.setRefreshing(false);
                }

                @Override
                public void onNext(AppInfo appInfo) {
                    mAddedApps.add(appInfo);
                    mAdapter.addApplication(mAddedApps.size() - 1
,appInfo);
                }
            });
}
```

正如你看到的，代码和之前的例子很像。这种方法让我们有机会来考虑一下代码的复用。

你可以将一个函数作为参数传给 `just()` 方法，你将会得到一个已存在代码的原始Observable版本。在一个新的响应式架构的基础上迁移已存在的代码，这个方法可能是一个有用的开始点。

repeat()

假如你想对一个Observable重复发射三次数据。例如，我们用 `just()` 例子中的Observable：

```
private void loadApps(AppInfo appOne,AppInfo appTwo,AppInfo appThree) {
    mRecyclerView.setVisibility(View.VISIBLE);
    Observable.just(appOne,appTwo,appThree)
            .repeat(3)
            .subscribe(new Observer<AppInfo>() {

                @Override
                public void onCompleted() {
                    mSwipeRefreshLayout.setRefreshing(false);
                    Toast.makeText(getActivity(), "Here is the list!", Toast.LENGTH_LONG).show();
                }

                @Override
                public void onError(Throwable e) {
                    Toast.makeText(getActivity(), "Something went wrong!", Toast.LENGTH_SHORT).show();
                    mSwipeRefreshLayout.setRefreshing(false);
                }

                @Override
                public void onNext(AppInfo appInfo) {
                    mAddedApps.add(appInfo);
                    mAdapter.addApplication(mAddedApps.size() - 1,appInfo);
                }
            });
}
```

正如你看到的，我们在 `just()` 创建Observable后追加了 `repeat(3)` ，它将会创建9个元素的序列，每一个都单独发射。

defer()

有这样一个场景，你想在这声明一个Observable但是你又想推迟这个Observable的创建直到观察者订阅时。看下面的 `getInt()` 函数：

```
private Observable<Integer> getInt(){
    return Observable.create(subscriber -> {
        if(subscriber.isUnsubscribed()){
            return;
        }
        App.L.debug("GETINT");
        subscriber.onNext(42);
        subscriber.onCompleted();
    });
}
```

这比较简单，并且它没有做太多事情，但是它正好为我们服务。现在，我们可以创建一个新的Observable并且应用 `defer()`：

```
Observable<Integer> deferred = Observable.defer(this::getInt);
```

这次， `deferred` 存在，但是 `getInt()` `create()` 方法还没有调用:logcat日志也没有“GETINT”打印出来:

```
deferred.subscribe(number -> {
    App.L.debug(String.valueOf(number));
});
```

但是一旦我们订阅了， `create()` 方法就会被调用并且我们也可以在logcat日志中得到下卖弄两个：GETINT和42。

range()

你需要从一个指定的数字X开始发射N个数字吗？你可以用 `range`：

```
Observable.range(10,3)
    .subscribe(new Observer<Integer>() {

        @Override
        public void onCompleted() {
            Toast.makeText(getActivity(), "Yeaaah!", Toast.LENGT
H_LONG).show();
        }

        @Override
        public void onError(Throwable e) {
            Toast.makeText(getActivity(), "Something went wrong!"
, Toast.LENGTH_SHORT).show();
        }

        @Override
        public void onNext(Integer number) {
            Toast.makeText(getActivity(), "I say " + number, Toa
st.LENGTH_SHORT).show();
        }
    });
```

`range()` 函数用两个数字作为参数：第一个是起始点，第二个是我们想发射数字的个数。

interval()

`interval()` 函数在你需要创建一个轮询程序时非常好用。

```
Subscription stopMePlease = Observable.interval(3,TimeUnit.SECON
DS)
    .subscribe(new Observer<Integer>() {

        @Override
        public void onCompleted() {
            Toast.makeText(getActivity(), "Yeaaah!", Toast.LENGT
H_LONG).show();
        }

        @Override
        public void onError(Throwable e) {
            Toast.makeText(getActivity(), "Something went wrong!"
, Toast.LENGTH_SHORT).show();
        }

        @Override
        public void onNext(Integer number) {
            Toast.makeText(getActivity(), "I say " + number, Toa
st.LENGTH_SHORT).show();
        }
    });
```

`interval()` 函数的两个参数：一个指定两次发射的时间间隔，另一个是用到的时间单位。

timer()

如果你需要一个一段时间之后才发射的Observable，你可以像下面的例子使用 `timer()` ：

```
Observable.timer(3,TimeUnit.SECONDS)
    .subscribe(new Observer<Long>() {

        @Override
        public void onCompleted() {

        }

        @Override
        public void onError(Throwable e) {

        }

        @Override
        public void onNext(Long number) {
            Log.d("RXJAVA", "I say " + number);
        }
    });
```

它将3秒后发射0,然后就完成了。让我们使用 `timer()` 的第三个参数，就像下面的例子：

```
Observable.timer(3,3,TimeUnit.SECONDS)
    .subscribe(new Observer<Long>() {

        @Override
        public void onCompleted() {

        }

        @Override
        public void onError(Throwable e) {

        }

        @Override
        public void onNext(Long number) {
            Log.d("RXJAVA", "I say " + number);
        }
    });
```

用这个代码，你可以创建一个以初始值来延迟（上一个例子是3秒）执行的 `interval()` 版本，然后每隔N秒就发射一个新的数字（前面的例子是3秒）。

总结

在本章中，我们创建了第一个由RxJava强化的Android应用程序。我们从头、从已有的列表、从已有的函数来创建Observable。我们也学习了如何创建重复发射的Observables，间隔发射的Observables以及延迟发射的Observables。

在下一章中，我们将掌握过滤操作，能够从我们接收到的序列中创建我们需要的序列。

过滤Observables

在上一章中，我们学习了使用RxJava创建一个Android工程以及如何创建一个可观测的列表来填充RecyclerView。我们现在知道了如何从头、从列表、从一个已存在的传统Java函数来创建Observable。

这一章中，我们将研究可观测序列的本质：过滤。我们将学到如何从发射的Observable中选取我们想要的值，如何获取有限个数的值，如何处理溢出的场景，以及更多的有用的技巧。

过滤序列

RxJava让我们使用 `filter()` 方法来过滤我们观测序列中不想要的值，在上一章中，我们在几个例子中使用了已安装的应用列表，但是我们只想展示以字母 `C` 开头的已安装的应用该怎么办呢？在这个新的例子中，我们将使用同样的列表，但是我们会过滤它，通过把合适的谓词传给 `filter()` 函数来得到我们想要的值。

上一章中 `loadList()` 函数可以改成这样：

```
private void loadList(List<AppInfo> apps) {
    mRecyclerView.setVisibility(View.VISIBLE);
    Observable.from(apps)
            .filter((appInfo) ->
            appInfo.getName().startsWith("C"))
            .subscribe(new Observer<AppInfo>() {

                @Override
                public void onCompleted() {
                    mSwipeRefreshLayout.setRefreshing(false);
                }

                @Override
                public void onError(Throwable e) {
                    Toast.makeText(getActivity(), "Something wen
t wrong!", Toast.LENGTH_SHORT).show();
                    mSwipeRefreshLayout.setRefreshing(false);
                }

                @Override
                public void onNext(AppInfo appInfo) {
                    mAddedApps.add(appInfo);
                    mAdapter.addApplication(mAddedApps.size() - 1
,appInfo);
                }
            });
}
```

我们从上一章中的 `loadList()` 函数中添加下面一行：

```
.fliter((appInfo -> appInfo.getName().startsWith("C"))
```

创建Observable完以后，我们从发出的每个元素中过滤掉开头字母不是C的。为了让这里更清楚一些，我们用Java 7的语法来实现：

```
.filter(new Func1<AppInfo,Boolean>(){
    @Override
    public Boolean call(AppInfo appInfo){
        return appInfo.getName().startsWith("C");
    }
})
```

我们传一个新的 `Func1` 对象给 `filter()` 函数，即只有一个参数的函数。 `Func1` 有一个 `AppInfo` 对象来作为它的参数类型并且返回 `Boolean` 对象。只要条件符合 `filter()` 函数就会返回 `true` 。此时，值会发射出去并且所有的观察者都会接收到。

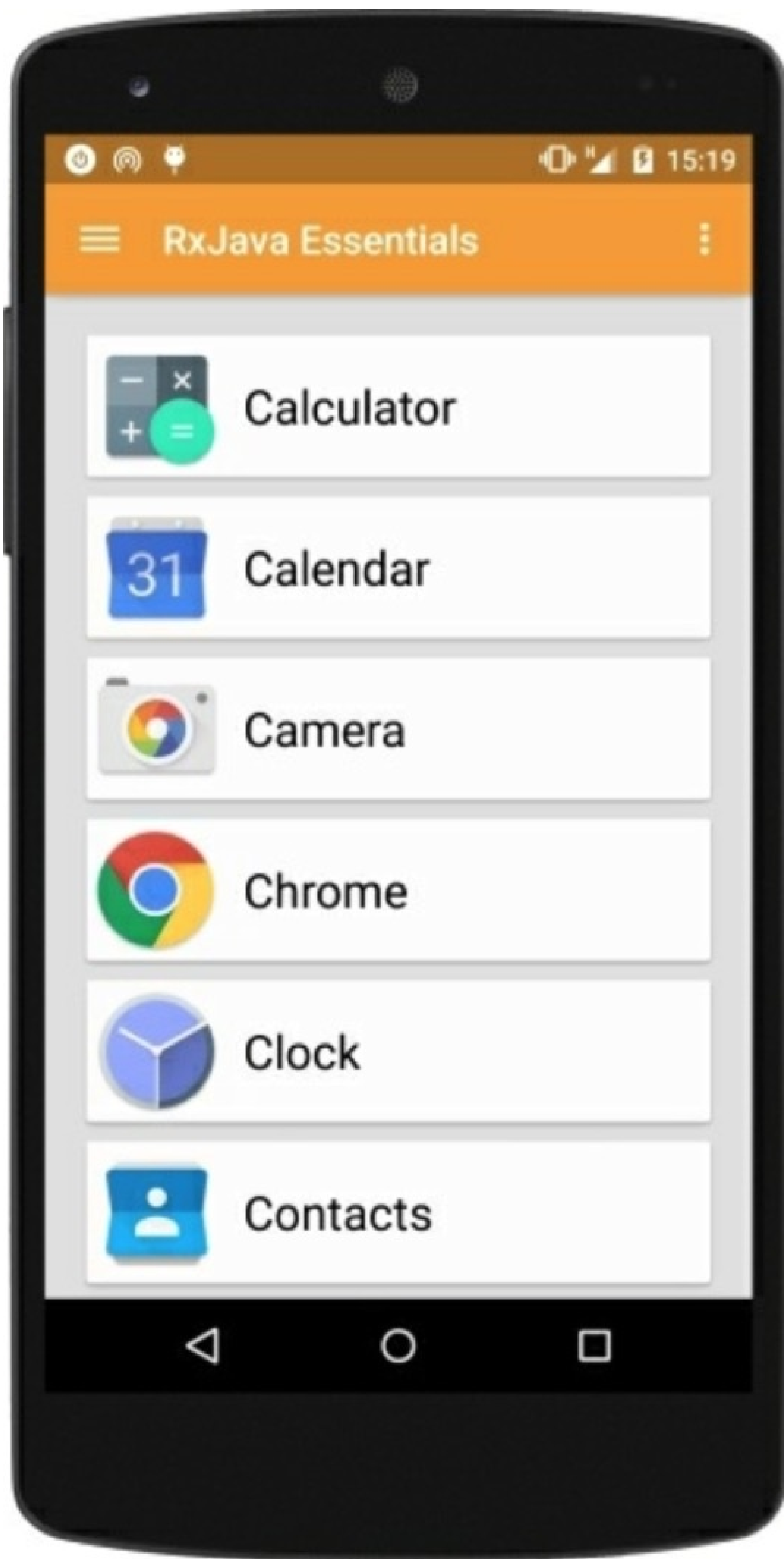
正如你想的那样，从一个我们得到的可观测序列中创建一个我们需要的序列 `filter()` 是很好用的。我们不需要知道可观测序列的源或者为什么发射这么多不同的数据。我们只是想要这些元素的子集来创建一个可以在应用中使用的新序列。这种思想促进了我们编码中的分离性与抽象性。

`filter()` 函数最常用的用法之一时过滤 `null` 对象：

```
.filter(new Func1<AppInfo,Boolean>(){
    @Override
    public Boolean call(AppInfo appInfo){
        return appInfo != null;
    }
})
```

这看起来简单，对于简单的事情有许多模板代码，但是它帮我们免去了在 `onNext()` 函数调用中再去检测 `null` 值，让我们把注意力集中在应用业务逻辑上。

下图展示了过滤出的C字母开头的已安装的应用列表。


15:19
RxJava Essentials
Calculator
31
Calendar
Camera
Chrome
Clock
Contacts

获取我们需要的数据

当我们不需要整个序列时，而是只想取开头或结尾的几个元素，我们可以用 `take()` 或 `takeLast()` 。

Take

如果我们只想要一个可观测序列中的前三个元素那将会怎么样，发射它们，然后让Observable完成吗？ `take()` 函数用整数N来作为一个参数，从原始的序列中发射前N个元素，然后完成：

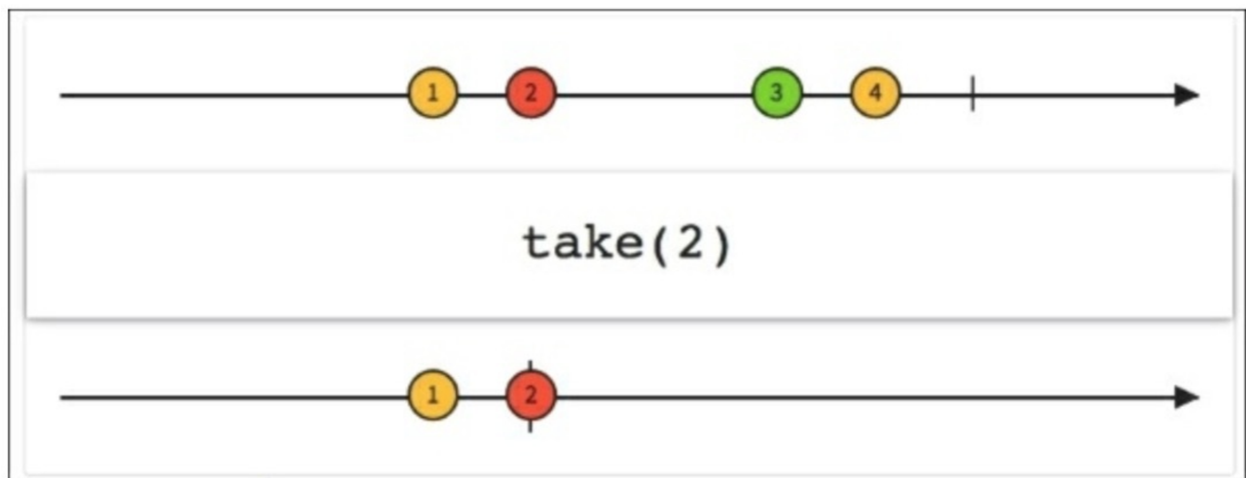
```
private void loadList(List<AppInfo> apps) {
    mRecyclerView.setVisibility(View.VISIBLE);
    Observable.from(apps)
            .take(3)
            .subscribe(new Observer<AppInfo>() {

                @Override
                public void onCompleted() {
                    mSwipeRefreshLayout.setRefreshing(false);
                }

                @Override
                public void onError(Throwable e) {
                    Toast.makeText(getActivity(), "Something wen
t wrong!", Toast.LENGTH_SHORT).show();
                    mSwipeRefreshLayout.setRefreshing(false);
                }

                @Override
                public void onNext(AppInfo appInfo) {
                    mAddedApps.add(appInfo);
                    mAdapter.addApplication(mAddedApps.size() - 1
,appInfo);
                }
            });
}
```

下图中展示了发射数字的一个可观测序列。我们对这个可观测序列应用 `take(2)` 函数，然后我们创建一个只发射可观测源的第一个和第二个数据的新序列。



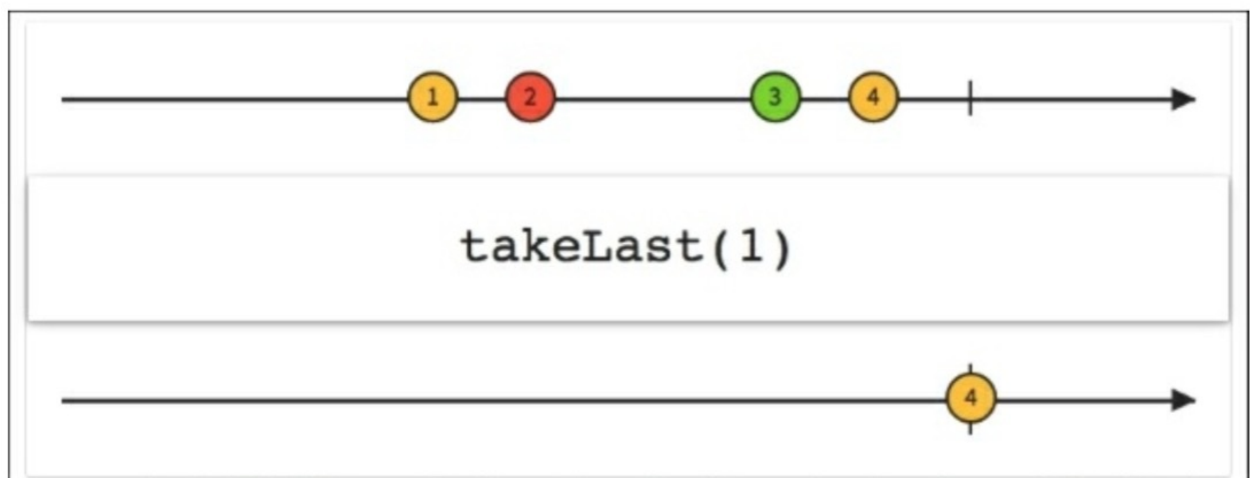
TakeLast

如果我们想要最后N个元素，我们只需使用 `takeLast()` 函数：

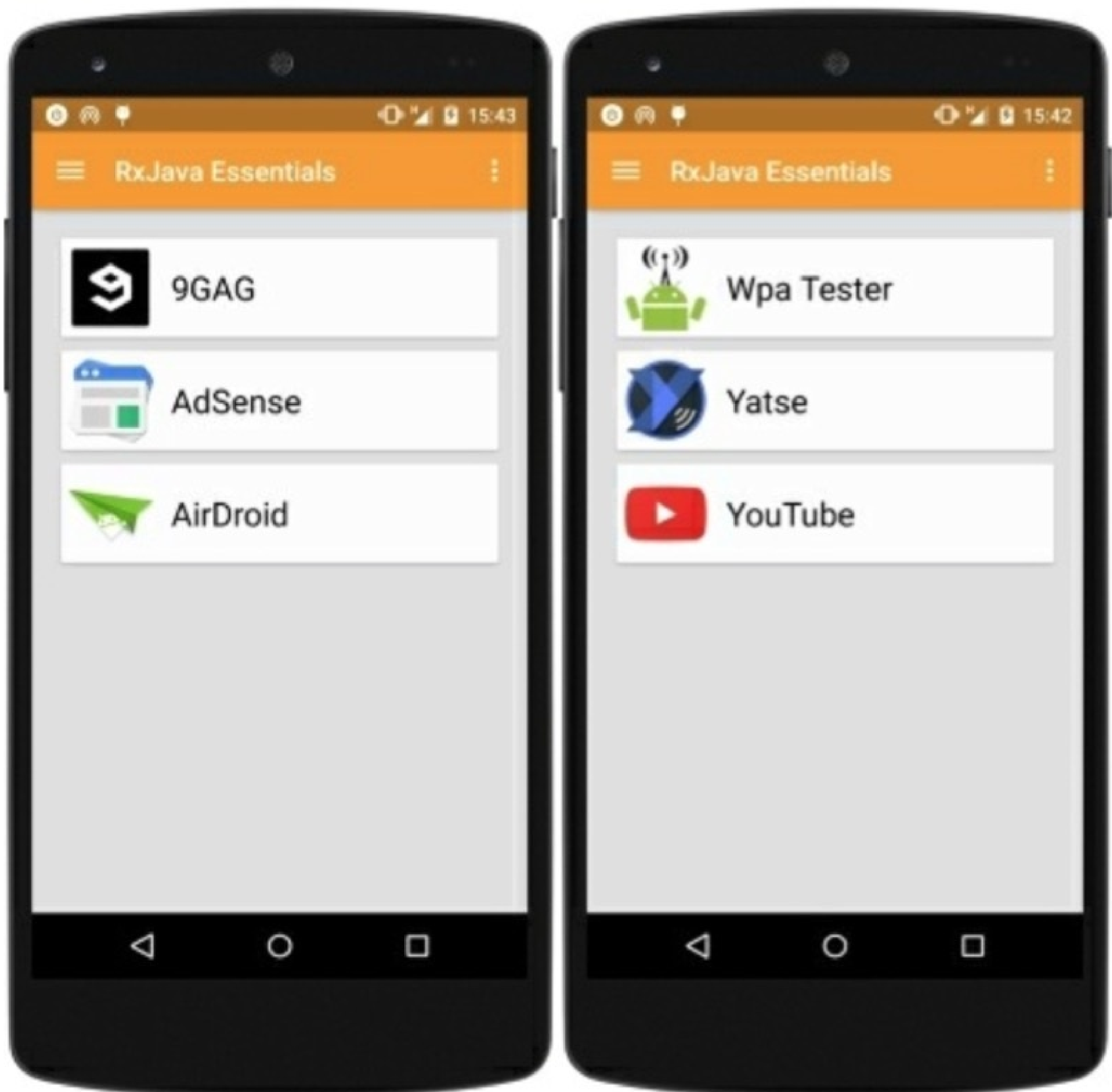
```
Observable.from(apps)
        .takeLast(3)
        .subscribe(...);
```

正如听起来那样不值一提，值得注意的是因为takeLast()方法只能作用于一组有限的序列（发射元素），它只能应用于一个完整的序列。

下图中展示了如何从可观测源中发射最后一个元素来创建一个新的序列：



下图中展示了我们在已安装的应用列表使用 `take()` 和 `takeLast()` 函数后发生的结果：


15:43
RxJava Essentials
9GAG
AdSense
AirDroid
15:42
RxJava Essentials
Wpa Tester
Yatse
YouTube

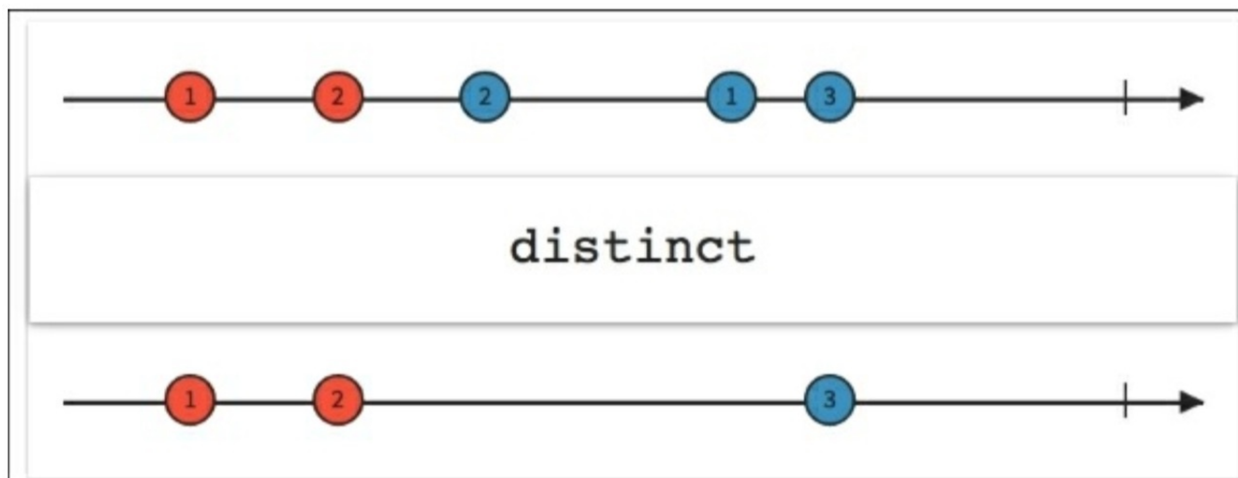
有且仅有一次

一个可观测序列会在出错时重复发射或者被设计成重复发射。`distinct()` 和 `distinctUntilChanged()` 函数可以方便的让我们处理这种重复问题。

Distinct

如果我们想对一个指定的值仅处理一次该怎么办？我们可以对我们的序列使用 `distinct()` 函数去掉重复的。就像 `takeLast()` 一样，`distinct()` 作用于一个完整的序列，然后得到重复的过滤项，它需要记录每一个发射的值。如果你在处理一大堆序列或者大的数据记得关注内存使用情况。

下图展示了如何在一个发射1和2两次的可观测源上创建一个无重的序列：



为了创建我们例子中序列，我们将使用我们至今已经学到的几个方法：

- `take()`：它有一小组的可识别的数据项。
- `repeat()`：创建一个有重复的大的序列。

然后，我们将应用 `distinct()` 函数来去除重复。

注意

我们用程序实现一个重复的序列，然后过滤出它们。这听起来时不可思议的，但是为了实现这个例子来使用我们至今为止已学习到的东西则是个不错的练习。

```
Observable<AppInfo> fullOfDuplicates = Observable.from(apps)
    .take(3)
    .repeat(3);
```

`fullOfDuplicates` 变量里把我们已安装应用的前三个重复了3次：有9个并且许多重复的。然后，我们使用 `distinct()`：

```
fullOfDuplicates.distinct()
            .subscribe(new Observer<AppInfo>() {

                @Override
                public void onCompleted() {
                    mSwipeRefreshLayout.setRefreshing(false);
                }

                @Override
                public void onError(Throwable e) {
                    Toast.makeText(getActivity(), "Something wen
t wrong!", Toast.LENGTH_SHORT).show();
                    mSwipeRefreshLayout.setRefreshing(false);
                }

                @Override
                public void onNext(AppInfo appInfo) {
                    mAddedApps.add(appInfo);
                    mAdapter.addApplication(mAddedApps.size() - 1
,appInfo);
                }
            });
}
```

结果，很明显，我们得到：


12:44
RxJava Essentials
9GAG
AdSense
AirDroid

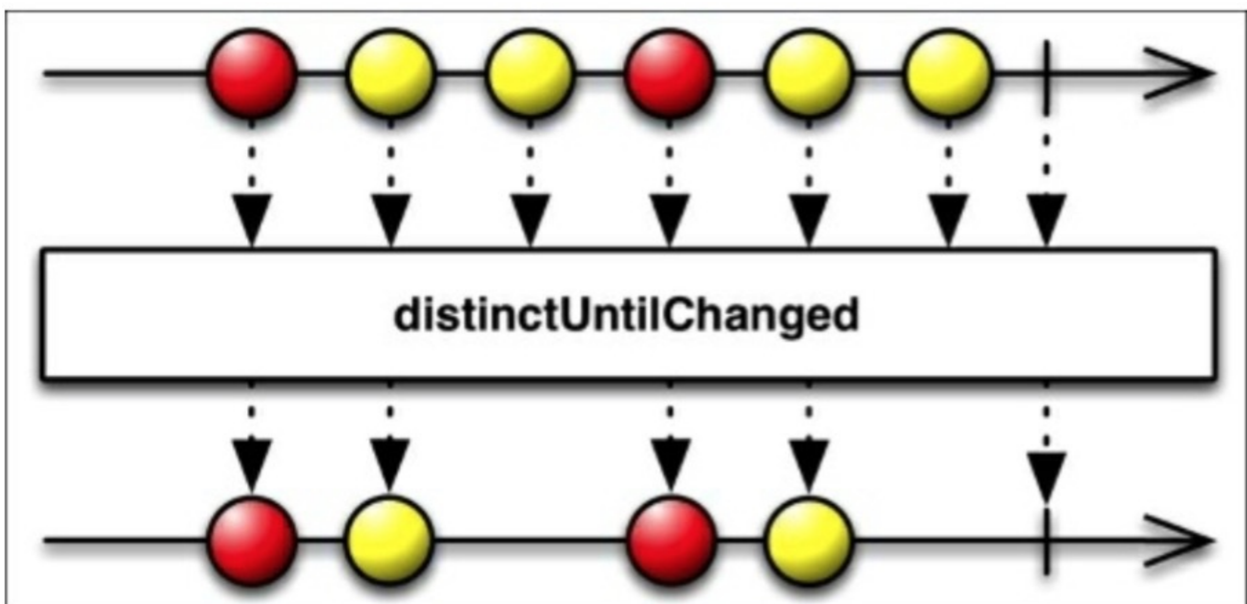
DistinctUntilsChanged

如果在一个可观测序列发射一个不同于之前的一个新值时让我们得到通知这时候该怎么做？我们猜想一下我们观测的温度传感器，每秒发射的室内温度：

```
21°...21°...21°...21°...22°...
```

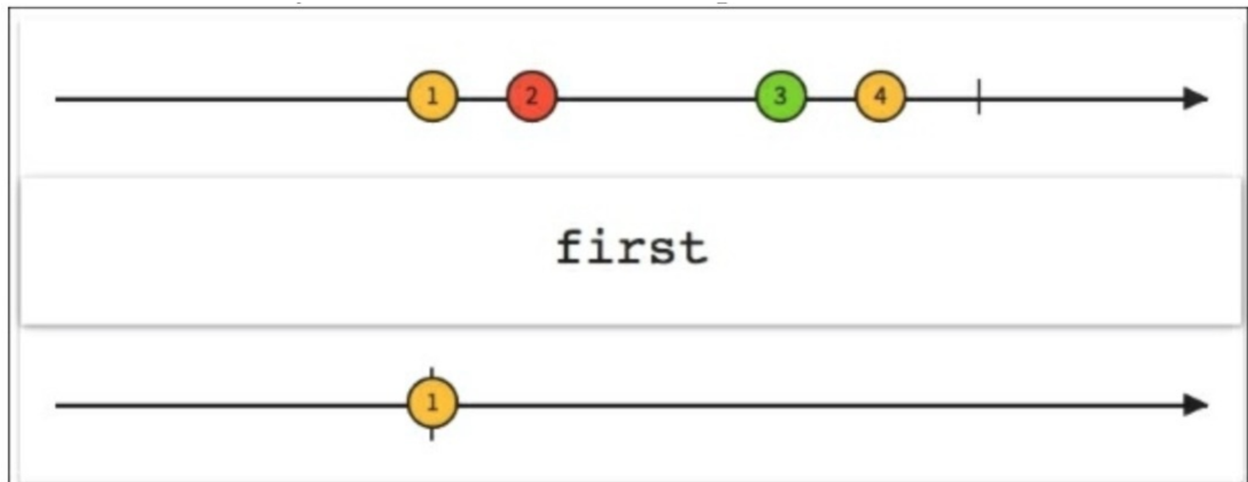
每次我们获得一个新值，我们都会更新当前正在显示的温度。我们出于系统资源保护并不想在每次值一样时更新数据。我们想忽略掉重复的值并且在温度确实改变时才想得到通知。`ditinctUntilChanged()`过滤函数能做到这一点。它能轻易的忽略掉所有的重复并且只发射出新的值。

下图用图形化的方式展示了我们如何将`distinctUntilChanged()`函数应用在一个存在的序列上来创建一个新的不重复发射元素的序列。



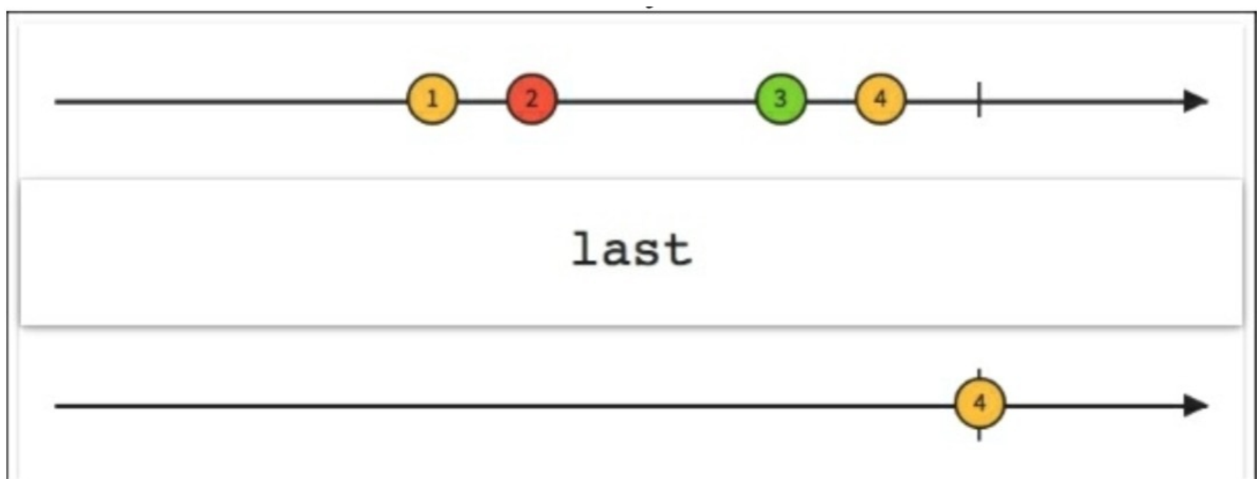
First and last

下图展示了如何从一个从可观测源序列中创建只发射第一个元素的序列。



`first()` 方法和 `last()` 方法很容易弄明白。它们从Observable中只发射第一个元素或者最后一个元素。这两个都可以传 `Func1` 作为参数，：一个可以确定我们感兴趣的第一个或者最后一个的谓词：

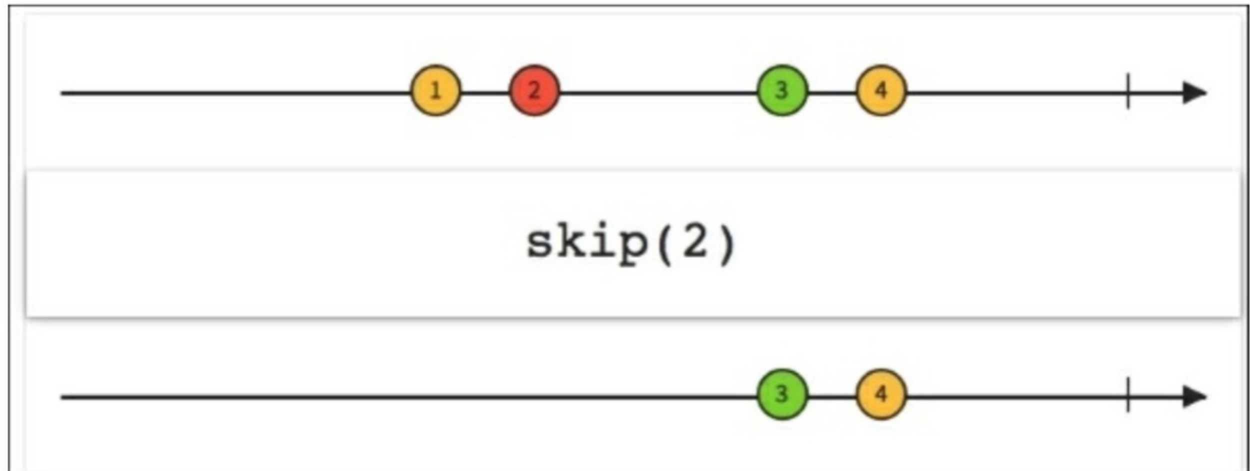
下图展示了 `last()` 应用在一个完成的序列上来创建一个仅仅发射最后一个元素的新的Observable。



与 `first()` 和 `last()` 相似的变量有： `firstOrDefault()` 和 `lastOrDefault()` .这两个函数当可观测序列完成时不再发射任何值时用得上。在这种场景下，如果Observable不再发射任何值时我们可以指定发射一个默认的值

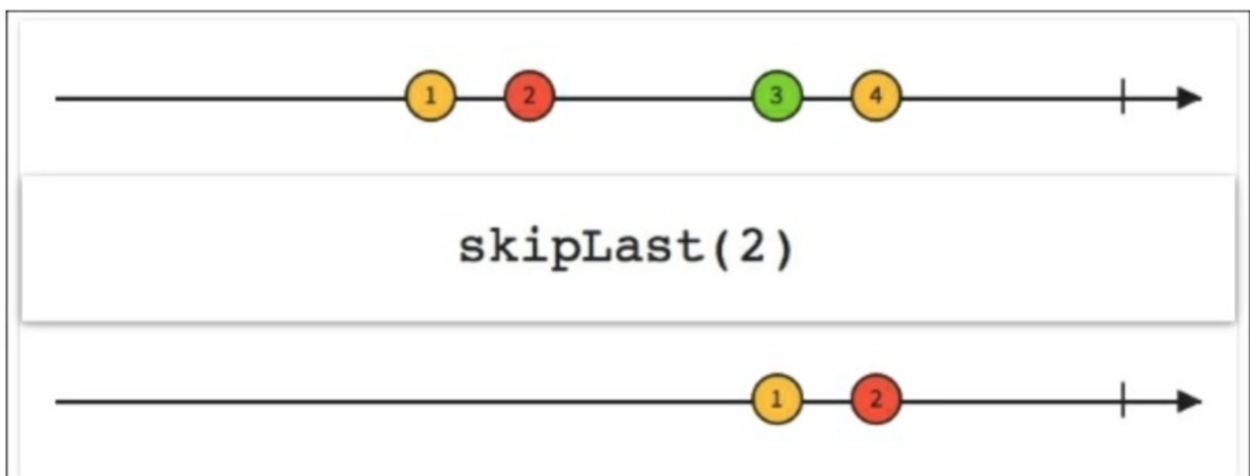
Skip and SkipLast

下图中展示了如何使用 `skip(2)` 来创建一个不发射前两个元素而是发射它后面的那些数据的序列。



`skip()` 和 `skipLast()` 函数与 `take()` 和 `takeLast()` 相对应。它们用整数N作参数，从本质上来说，它们不让Observable发射前N个或者后N个值。如果我们知道一个序列以没有太多用的“可控”元素开头或结尾时我们可以使用它。

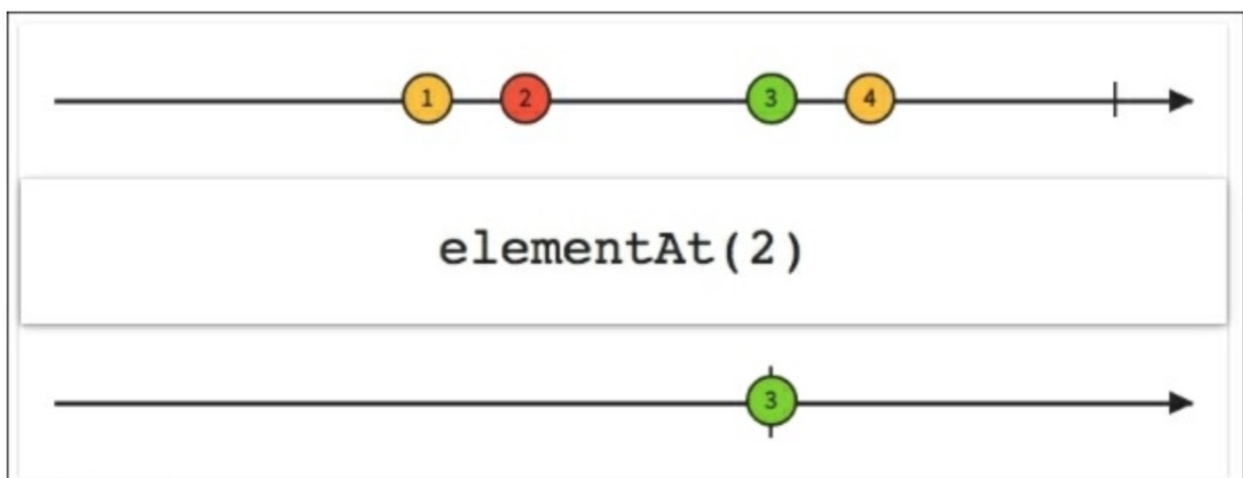
下图与前一个场景相对应：我们创建一个新的序列，它会跳过后面两个元素从源序列中发射剩下的其他元素。



ElementAt

如果我们只想要可观测序列发射的第五个元素该怎么办？ `elementAt()` 函数仅从一个序列中发射第n个元素然后就完成了。

如果我们想查找第五个元素但是可观测序列只有三个元素可供发射时该怎么办？我们可以使用 `elementAtOrDefault()` 。下图展示了如何通过使用 `elementAt(2)` 从一个序列中选择第三个元素以及如何创建一个只发射指定元素的新的Observable。



Sampling

让我们再回到那个温度传感器。它每秒都会发射当前室内的温度。说实话，我们并不认为温度会变化这么快，我们可以使用一个小的发射间隔。在Observable后面加一个 `sample()` ，我们将创建一个新的可观测序列，它将在一个指定的时间间隔里由Observable发射最近一次的数值：

```
Observable<Integer> sensor = [...]

sensor.sample(30,TimeUnit.SECONDS)
    .subscribe(new Observer<Integer>() {

        @Override
        public void onCompleted() {

        }

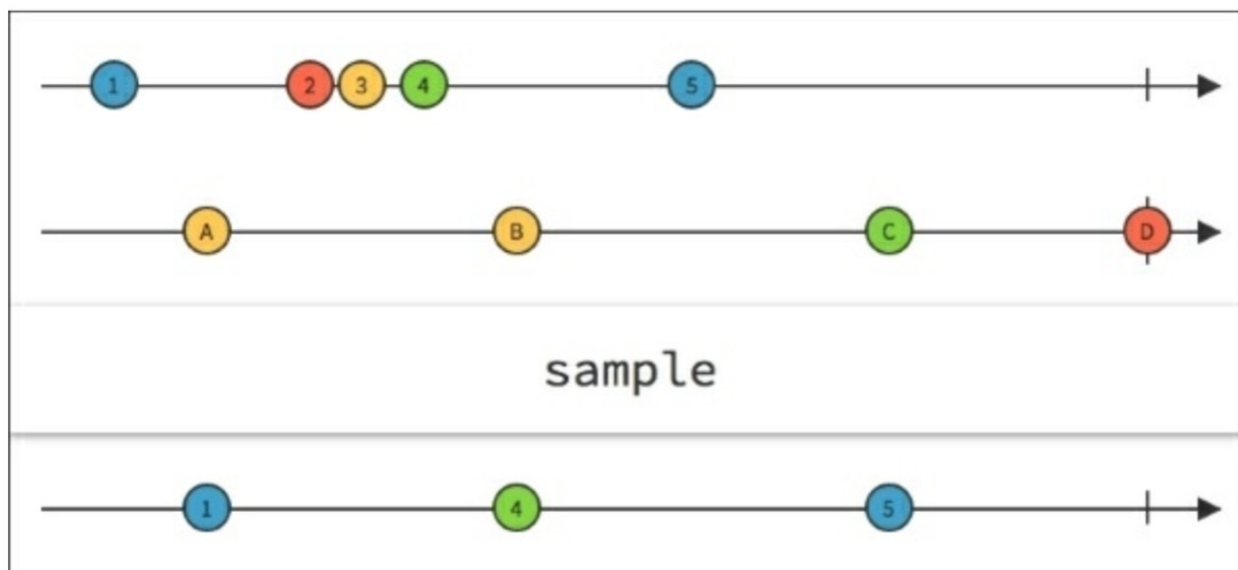
        @Override
        public void onError(Throwable e) {

        }

        @Override
        public void onNext(Integer currentTemperature) {
            updateDisplay(currentTemperature)
        }
    });
```

例子中Observable将会观测温度Observable然后每隔30秒就会发射最后一个温度值。很明显， `sample()` 支持全部的时间单位：秒，毫秒，天，分等等。

下图中展示了一个间隔发射字母的Observable如何采样一个发射数字的Observable。Observable的结果将会发射每个已发射字母的最后一组数据：1，4，5.



如果我们想让它定时发射第一个元素而不是最近的一个元素，我们可以使用 `throttleFirst()` 。

Timeout

假设我们工作的是一个时效性的环境，我们温度传感器每秒都在发射一个温度值。我们想让它每隔两秒至少发射一个，我们可以使用 `timeout()` 函数来监听源可观测序列,就是在我们设定的时间间隔内如果没有得到一个值则发射一个错误。我们可以认为 `timeout()` 为一个Observable的限时的副本。如果在指定的时间间隔内Observable不发射值的话，它监听的原始的Observable时就会触发 `onError()` 函数。

```
Subscription subscription = getCurrentTemperature()
    .timeout(2,TimeUnit.SECONDS)
    .subscribe(new Observer<Integer>() {

        @Override
        public void onCompleted() {

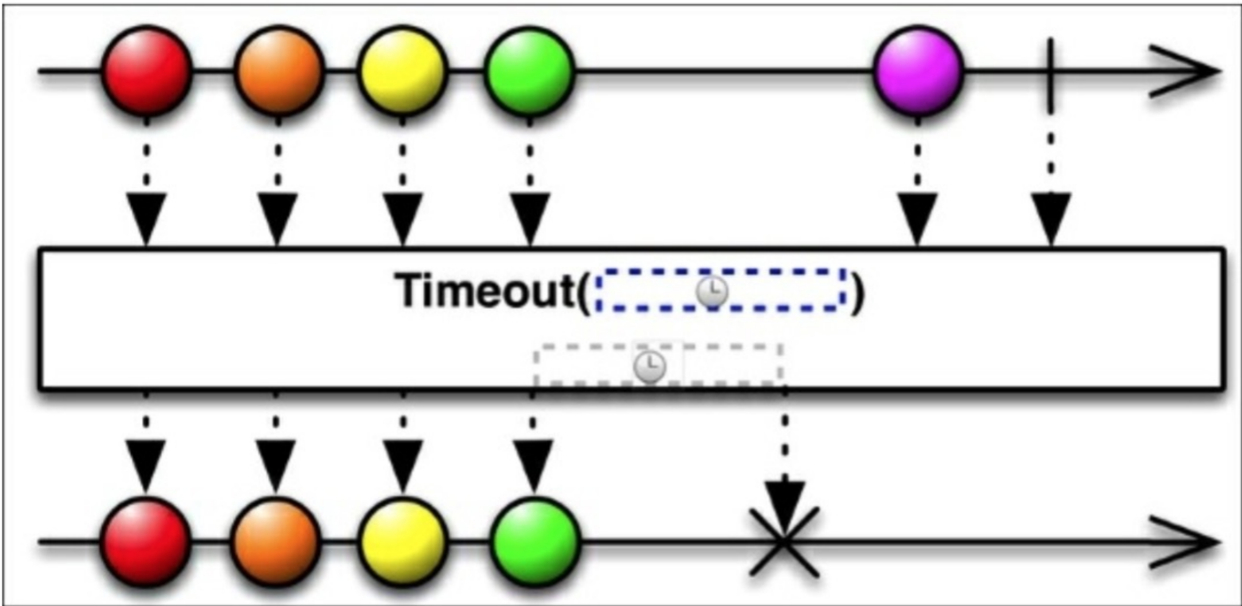
        }

        @Override
        public void onError(Throwable e) {
            Log.d("RXJAVA","You should go check the sensor, dude"
);
        }

        @Override
        public void onNext(Integer currentTemperature) {
            updateDisplay(currentTemperature)
        }
    });
```

和 `sample()` 一样， `timeout()` 使用 `TimeUnit` 对象来指定时间间隔。

下图中展示了一旦Observable超过了限时就会触发 `onError()` 函数：因为超时后它才到达，所以最后一个元素将不会发射出去。

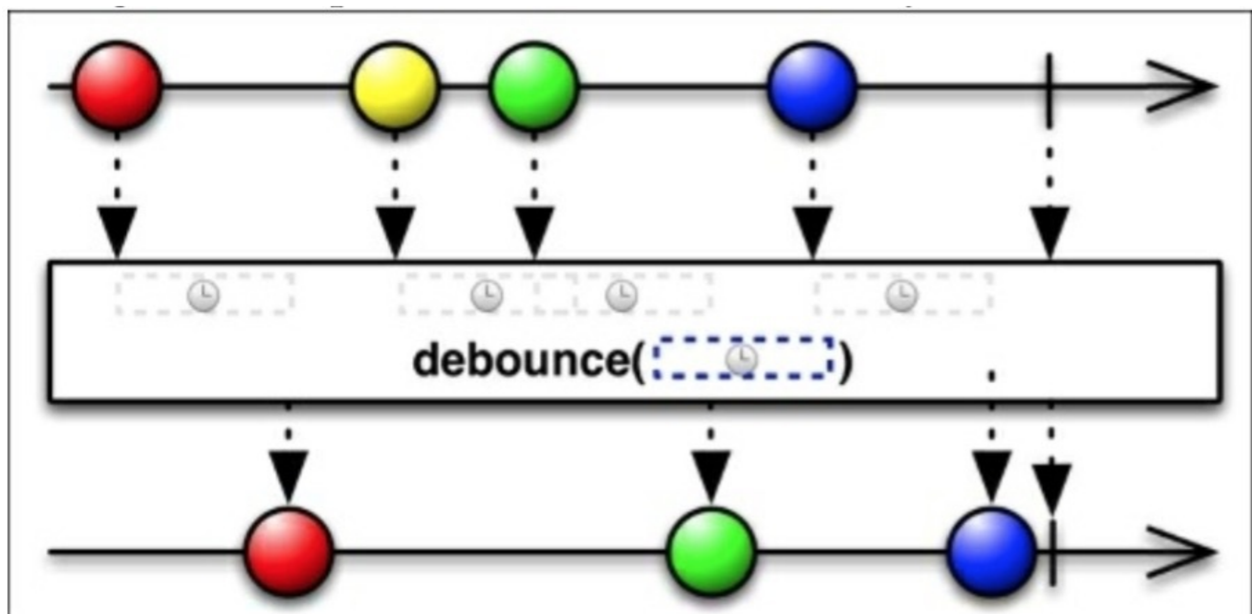

Timeout( )

Debounce

`debounce()` 函数过滤掉由Observable发射的速率过快的数据；如果在一个指定的时间间隔过去了仍旧没有发射一个，那么它将发射最后的那个。

就像 `sample()` 和 `timeout()` 函数一样， `debounce()` 使用 `TimeUnit` 对象指定时间间隔。

下图展示了多久从Observable发射一次新的数据， `debounce()` 函数开启一个内部定时器，如果在这个时间间隔内没有新的数据发射，则新的Observable发射出最后一个数据：



总结

这一章中，我们学习了如何过滤一个可观测序列。我们现在可以使用 `filter()` ， `skip()` ，和 `sample()` 来创建我们想要的Observable。

下一章中，我们将学习如何转换一个序列，将函数应用到每个元素，给它们分组和扫描来创建我们所需要的能完成目标的特定Observable。

转换Observables

在上一章中，我们探索了RxJava通用过滤方法。我们学习了如何使用 `filter()` 方法过滤我们不需要的值，如何使用 `take()` 得到发射元素的子集，如何使用 `distinct()` 函数来去除重复的。我们学习了如何借助 `timeout()` ， `sample()` ，以及 `debounce()` 来利用时间。

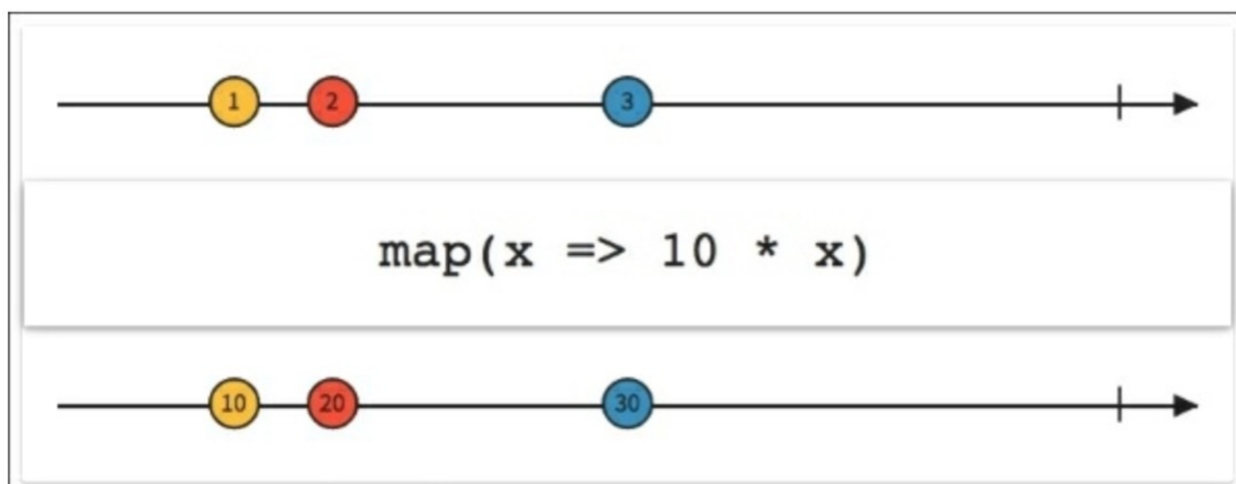
这一章中，我们将通过学习如何变换可观测序列来创建一个能够更好的满足我们需求的序列。

*map家族

RxJava提供了几个mapping函数： `map()` , `flatMap()` , `concatMap()` , `flatMapIterable()` 以及 `switchMap()` .所有这些函数都作用于一个可观测序列，然后变换它发射的值，最后用一种新的形式返回它们。让我们用合适的“真实世界”的例子一个个的学习下。

Map

RxJava的 `map` 函数接收一个指定的 `Func` 对象然后将它应用到每一个由Observable发射的值上。下图展示了如何将一个乘法函数应用到每个发出的值上以此创建一个新的Observable来发射转换的数据。



考虑下我们已安装的应用列表。我们怎么才能够显示同样的列表，但是又要所有的名字都小写呢？

我们的 `loadList()` 函数可以改成这样：

```
private void loadList(List<AppInfo> apps) {
    mRecyclerView.setVisibility(View.VISIBLE);
    Observable.from(apps)
        .map(new Func1<AppInfo,AppInfo>(){
            @Override
            public Appinfo call(AppInfo appInfo){
                String currentName = appInfo.getName();
                String lowerCaseName = currentName.toLowerCase()
;
                appInfo.setName(lowerCaseName);
                return appInfo;
            }
        })
        .subscribe(new Observer<AppInfo>() {

            @Override
            public void onCompleted() {
                mSwipeRefreshLayout.setRefreshing(false);
            }

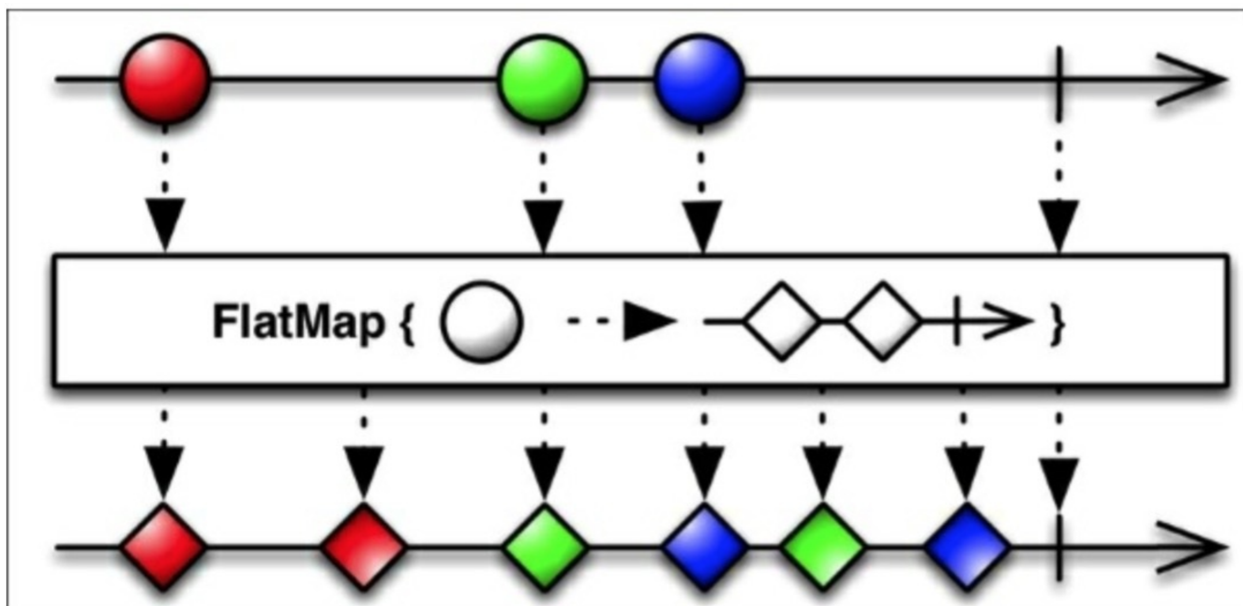
            @Override
            public void onError(Throwable e) {
                Toast.makeText(getActivity(), "Something went wr
ong!", Toast.LENGTH_SHORT).show();
                mSwipeRefreshLayout.setRefreshing(false);
            }

            @Override
            public void onNext(AppInfo appInfo) {
                mAddedApps.add(appInfo);
                mAdapter.addApplication(mAddedApps.size() - 1,ap
pInfo);
            }
        });
}
```

正如你看到的，像往常一样创建我们发射的Observable之后，我们追加一个 `map` 调用，我们创建一个简单的函数来更新 `AppInfo` 对象并提供一个名字小写的新版本给观察者。

FlatMap

在复杂的场景中，我们有一个这样的Observable：它发射一个数据序列，这些数据本身也可以发射Observable。RxJava的 `flatMap()` 函数提供一种铺平序列的方式，然后合并这些Observables发射的数据，最后将合并后的结果作为最终的Observable。

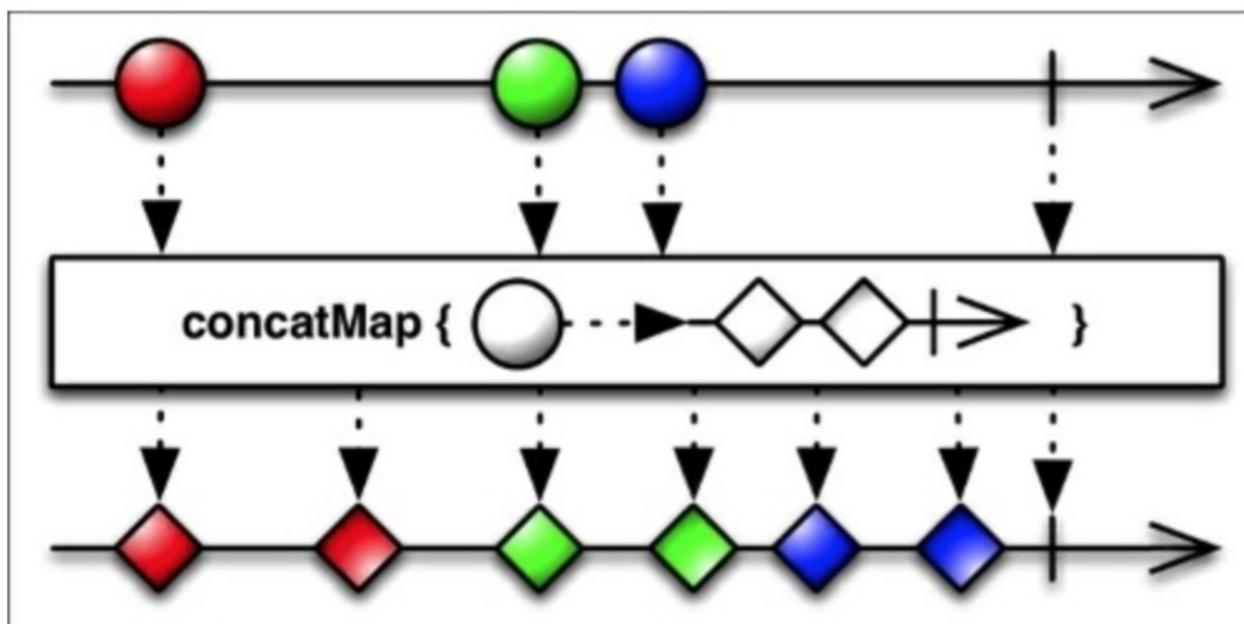


当我们在处理可能有大量的Observables时，重要是记住任何一个Observables发生错误的情况， `flatMap()` 将会触发它自己的 `onError()` 函数并放弃整个链。

重要的一点提示是关于合并部分：它允许交叉。正如上图所示，这意味着 `flatMap()` 不能够保证在最终生成的Observable中源Observables确切的发射顺序。

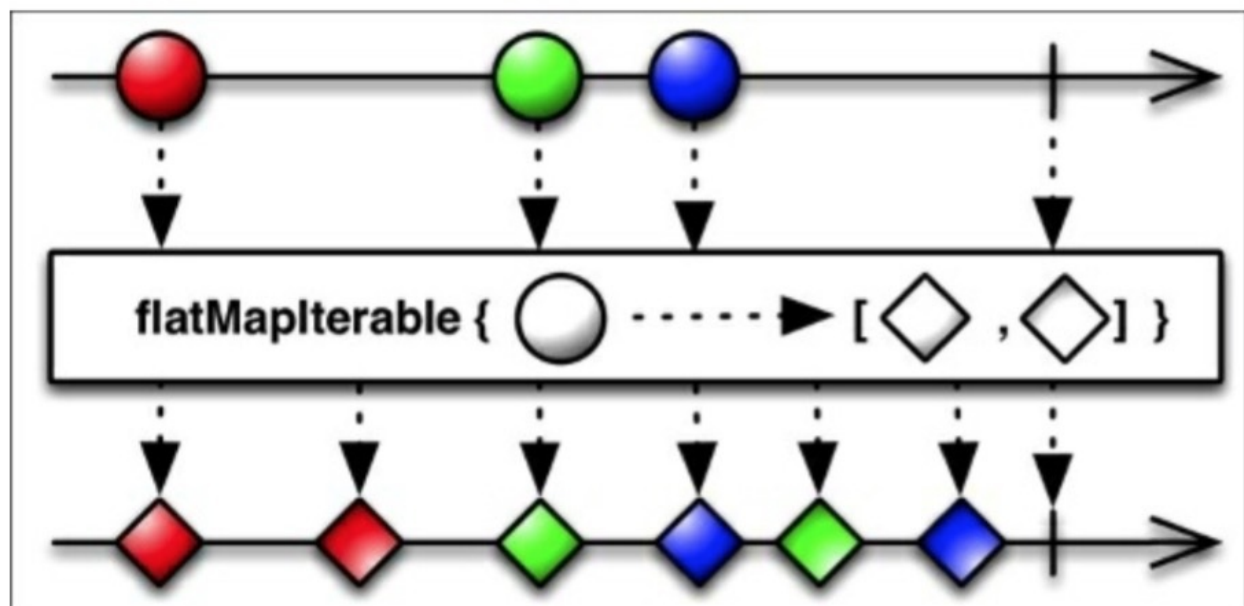
ConcatMap

RxJava的 `concatMap()` 函数解决了 `flatMap()` 的交叉问题，提供了一种能够把发射的值连续在一起的铺平函数，而不是合并它们，如下图所示：



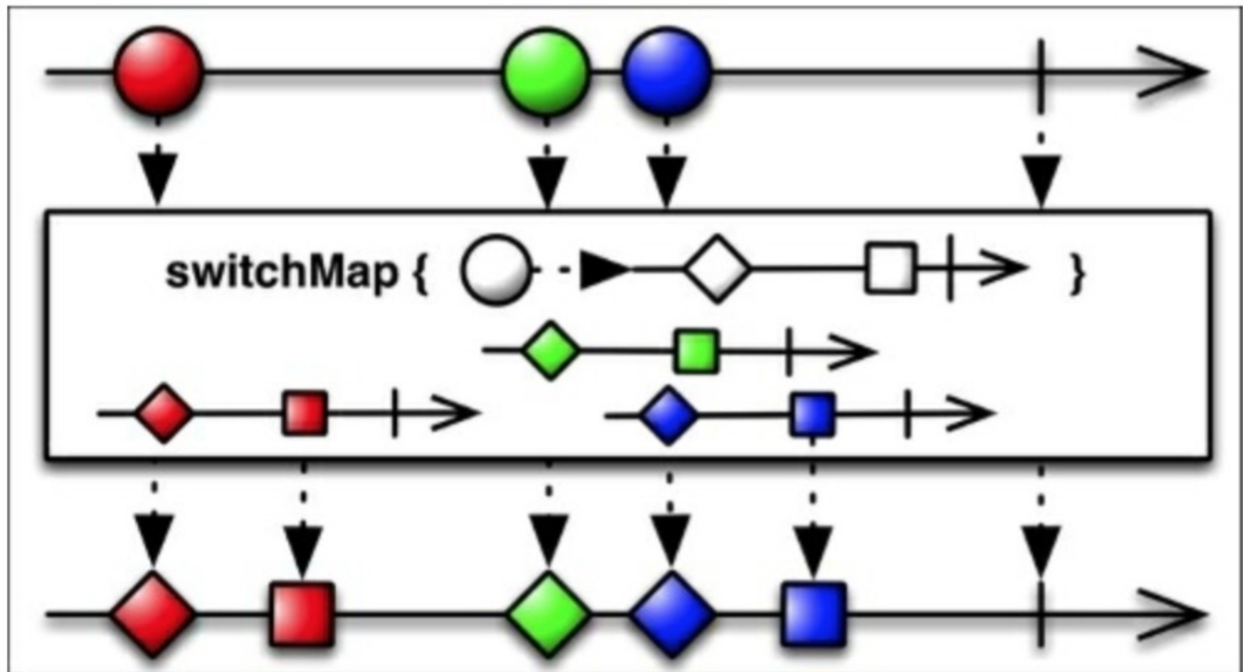
FlatMapIterable

作为*map家族的一员， `flatMapInterable()` 和 `flatMap()` 很像。仅有的本质不同是它将源数据两两结成对并生成Iterable，而不是原始数据项和生成的Observables。



SwitchMap

如下图所示， `switchMap()` 和 `flatMap()` 很像，除了一点：每当源Observable发射一个新的数据项（Observable）时，它将取消订阅并停止监视之前那个数据项产生的Observable，并开始监视当前发射的这一个。



Scan

RxJava的 `scan()` 函数可以看做是一个累积函数。 `scan()` 函数对原始Observable发射的每一项数据都应用一个函数，计算出函数的结果值，并将该值填充回可观测序列，等待和下一次发射的数据一起使用。

作为一个通用的例子，给出一个累加器：

```
Observable.just(1,2,3,4,5)
        .scan((sum,item) -> sum + item)
        .subscribe(new Subscriber<Integer>() {
            @Override
            public void onCompleted() {
                Log.d("RXJAVA", "Sequence completed.");
            }

            @Override
            public void onError(Throwable e) {
                Log.e("RXJAVA", "Something went south!");
            }

            @Override
            public void onNext(Integer item) {
                Log.d("RXJAVA", "item is: " + item);
            }
        });
```

我们得到的结果是：

```
RXJAVA: item is: 1
RXJAVA: item is: 3
RXJAVA: item is: 6
RXJAVA: item is: 10
RXJAVA: item is: 15
RXJAVA: Sequence completed.
```

我们也可以创建一个新版本的 `loadList()` 函数用来比较每个安装应用的名字从而创建一个名字长度递增的列表。

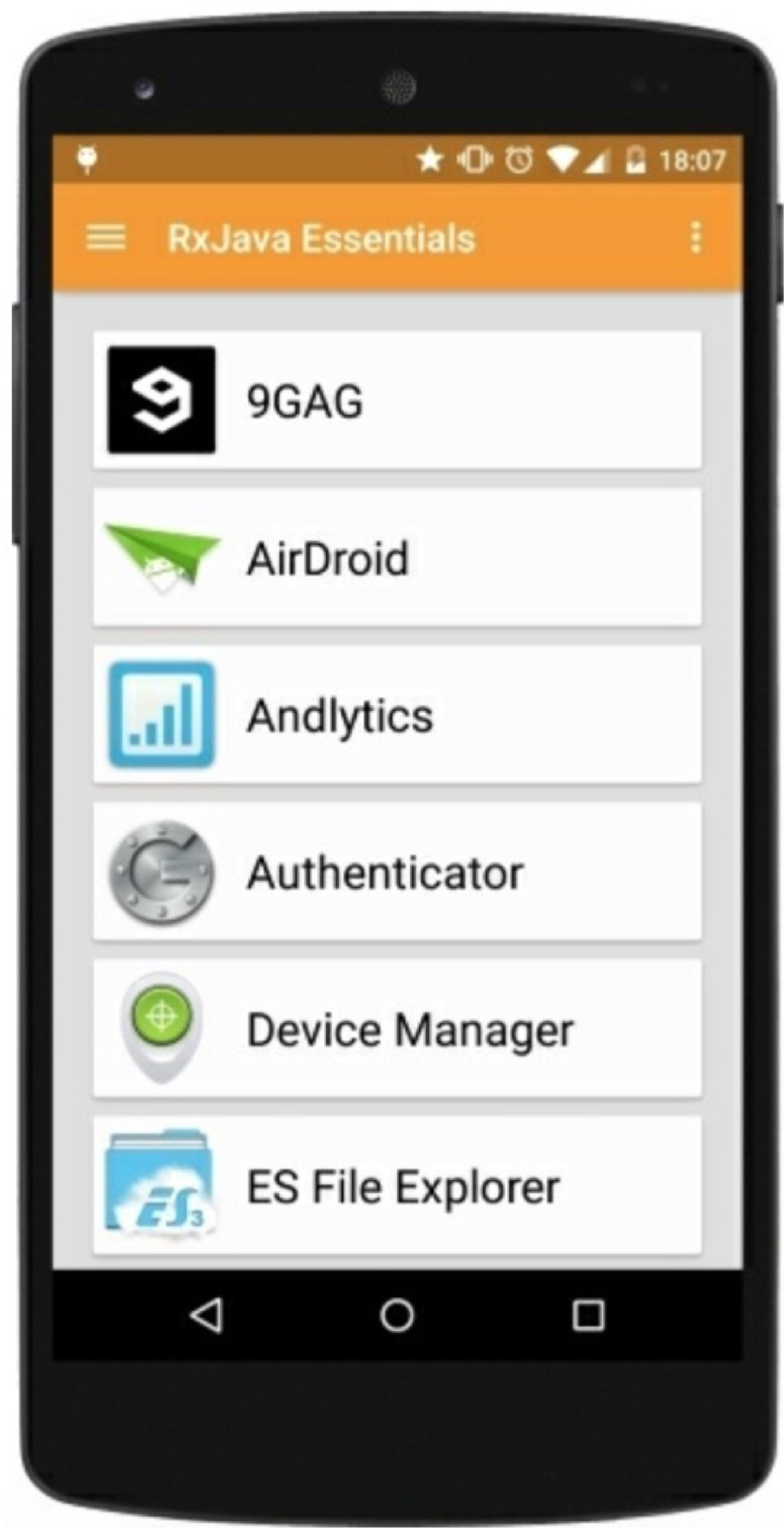
```
private void loadList(List<AppInfo> apps) {
    mRecyclerView.setVisibility(View.VISIBLE);
    Observable.from(apps)
            .scan((appInfo,appInfo2) -> {
                if(appInfo.getName().length > appInfo2.getName()
.length()){
                    return appInfo;
                } else {
                    return appInfo2;
                }
            })
            .distinct()
            .subscribe(new Observer<AppInfo>() {

                @Override
                public void onCompleted() {
                    mSwipeRefreshLayout.setRefreshing(false);
                }

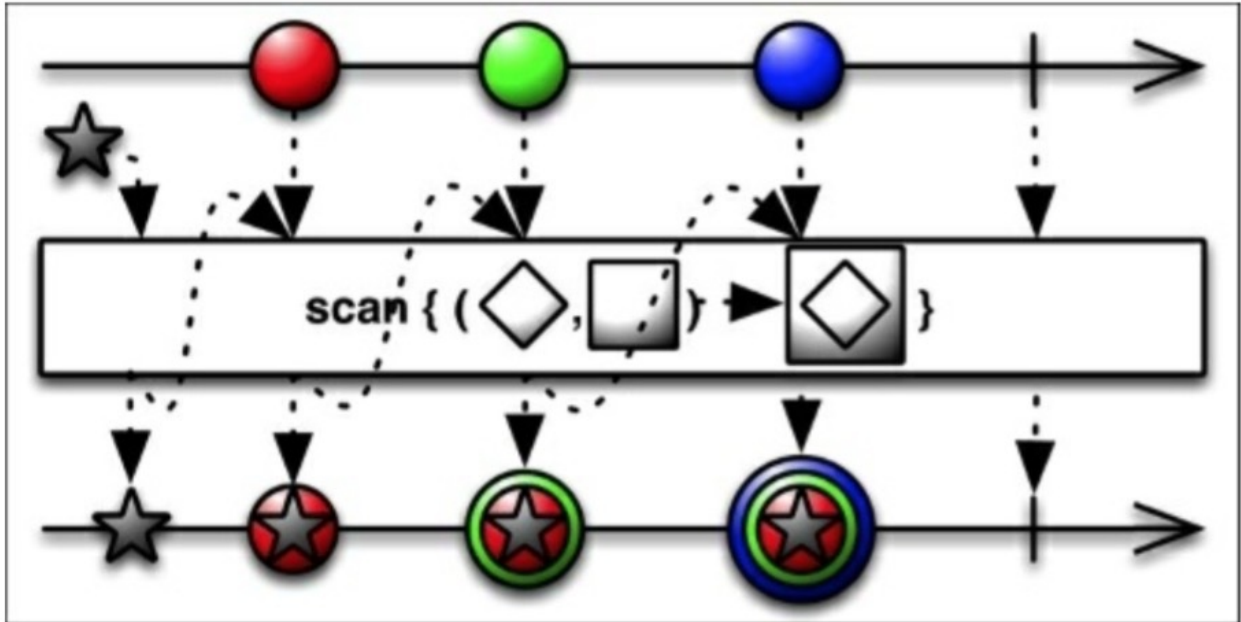
                @Override
                public void onError(Throwable e) {
                    Toast.makeText(getActivity(), "Something wen
t wrong!", Toast.LENGTH_SHORT).show();
                    mSwipeRefreshLayout.setRefreshing(false);
                }

                @Override
                public void onNext(AppInfo appInfo) {
                    mAddedApps.add(appInfo);
                    mAdapter.addApplication(mAddedApps.size() - 1
,appInfo);
                }
            });
}
```

结果如下：

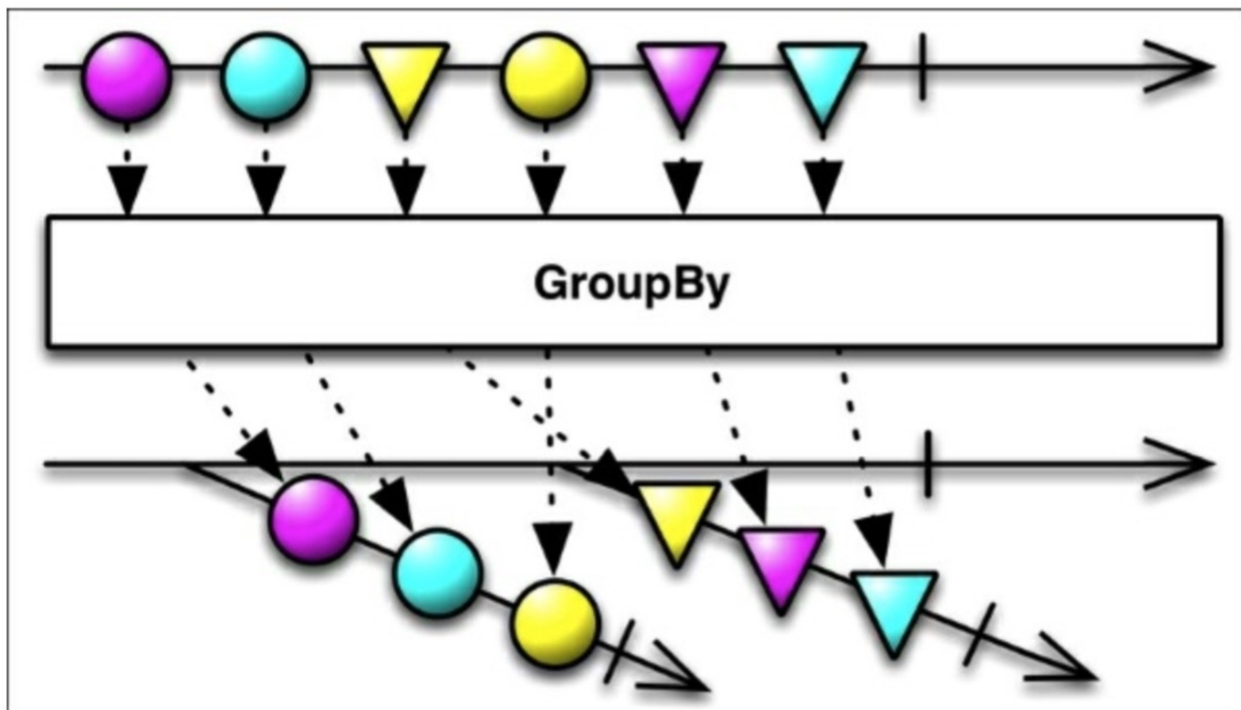

18:07
RxJava Essentials
9GAG
AirDroid
Andlytics
Authenticator
Device Manager
ES File Explorer

有一个 `scan()` 函数的变体，它用初始值作为第一个发射的值，方法签名看起来像： `scan(R,Func2)` ，如下图中的例子这样：



GroupBy

拿第一个例子开始，我们安装的应用程序列表按照字母表的顺序排序。然而，如果现在我们想按照最近更新日期来排序我们的App时该怎么办？RxJava提供了一个有用的函数从列表中按照指定的规则：`groupBy()` 来分组元素。下图中的例子展示了 `groupBy()` 如何将发射的值根据他们的形状来进行分组。



这个函数将源Observable变换成一个发射Observables的新的Observable。它们中的每一个新的Observable都发射一组指定的数据。

为了创建一个分组了的已安装应用列表，我们在 `loadList()` 函数中引入了一个新的元素：

```
Observable<GroupedObservable<String,AppInfo>> groupedItems = Obs
ervable.from(apps)
    .groupBy(new Func1<AppInfo,String>(){
        @Override
        public String call(AppInfo appInfo){
            SimpleDateFormat formatter = new SimpleDateFormat("M
M/yyyy");
            return formatter.format(new Date(appInfo.getLastUpda
teTime()));
        }
    });
```

现在我们创建了一个新的Observable，`groupedItems`，它将会发射一个带有`GroupedObservable`的序列。`GroupedObservable`是一个特殊的Observable，它源自一个分组的key。在这个例子中，key就是`String`，代表的意思是`Month/Year`格式化的最近更新日期。

这一点，我们已经创建了几个发射`AppInfo`数据的Observable，用来填充我们的列表。我们想保留字母排序和分组排序。我们将创建一个新的Observable将所有的联系起来，像往常一样然后订阅它：

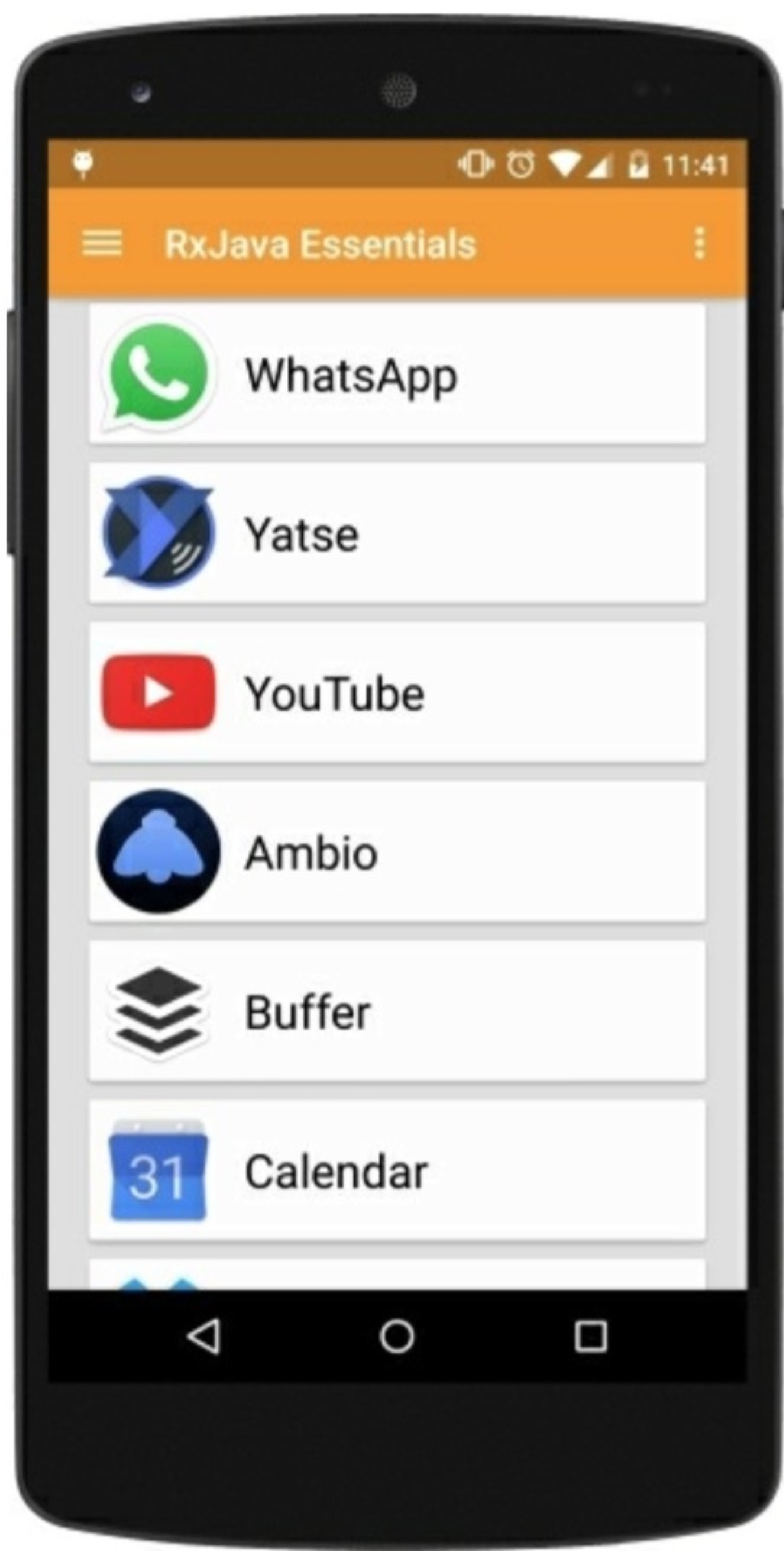
```
Observable.concat(groupedItems)
    .subscribe(new Observer<AppInfo>() {

        @Override
        public void onCompleted() {
            mSwipeRefreshLayout.setRefreshing(false);
        }

        @Override
        public void onError(Throwable e) {
            Toast.makeText(getActivity(), "Something went wrong!"
, Toast.LENGTH_SHORT).show();
            mSwipeRefreshLayout.setRefreshing(false);
        }

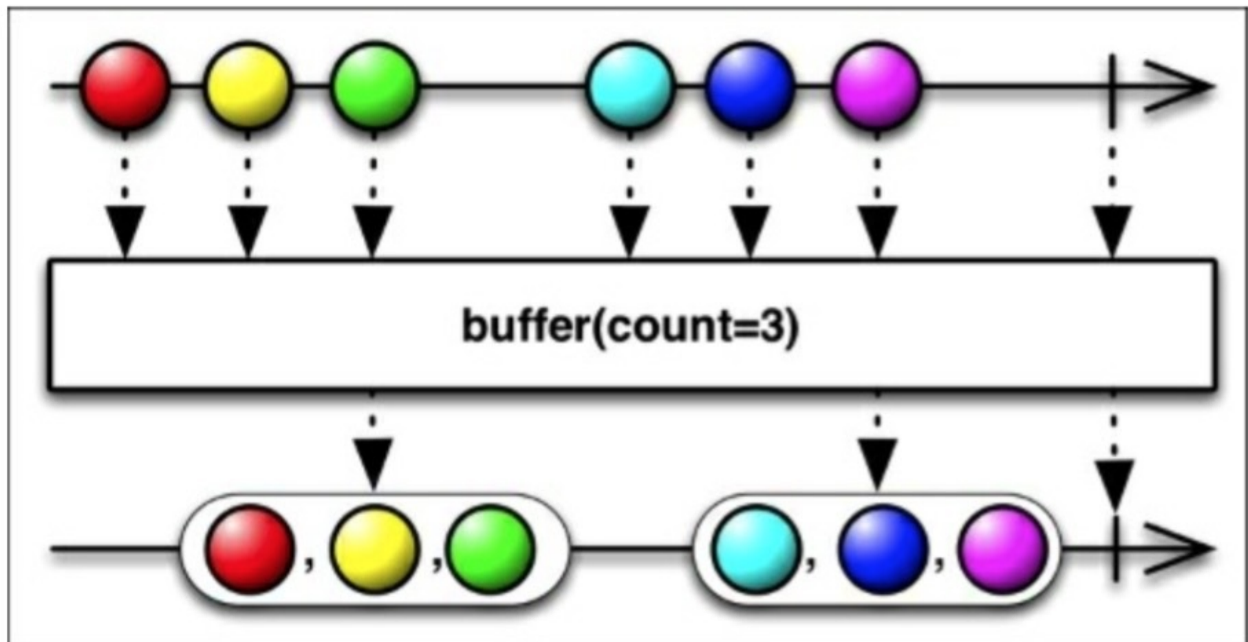
        @Override
        public void onNext(AppInfo appInfo) {
            mAddedApps.add(appInfo);
            mAdapter.addApplication(mAddedApps.size() - 1,appInf
o);
        }
    });
```

我们的 `loadList()` 函数完成了，结果是：

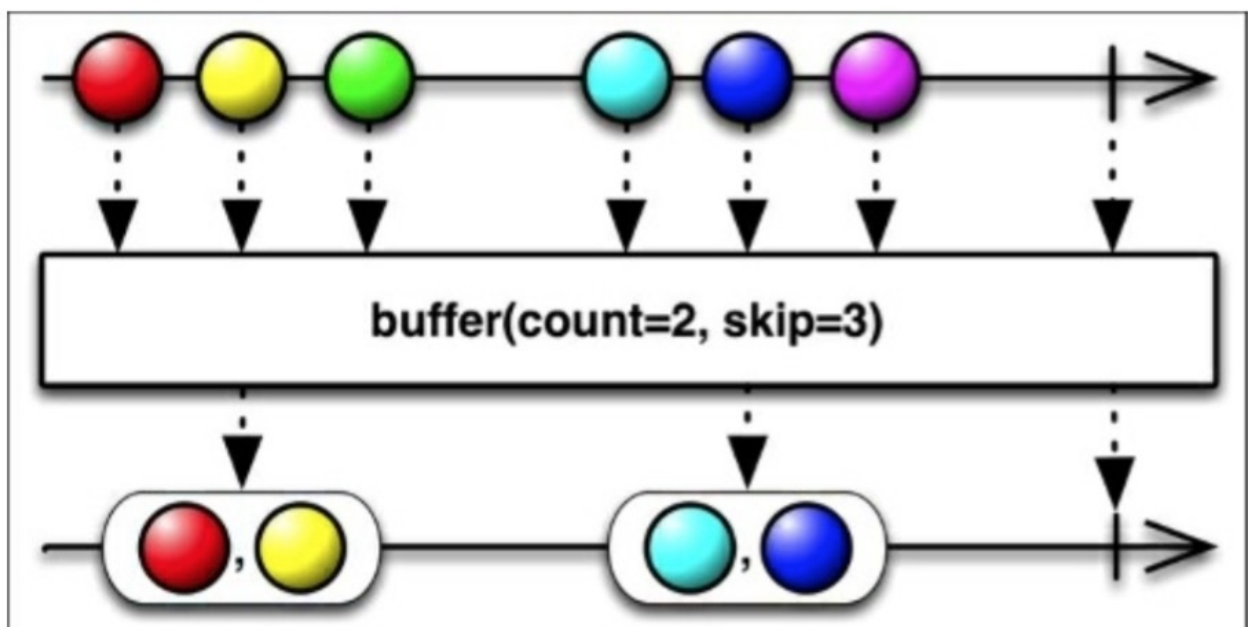

11:41
RxJava Essentials
WhatsApp
Yatse
YouTube
Ambio
Buffer
31
Calendar

Buffer

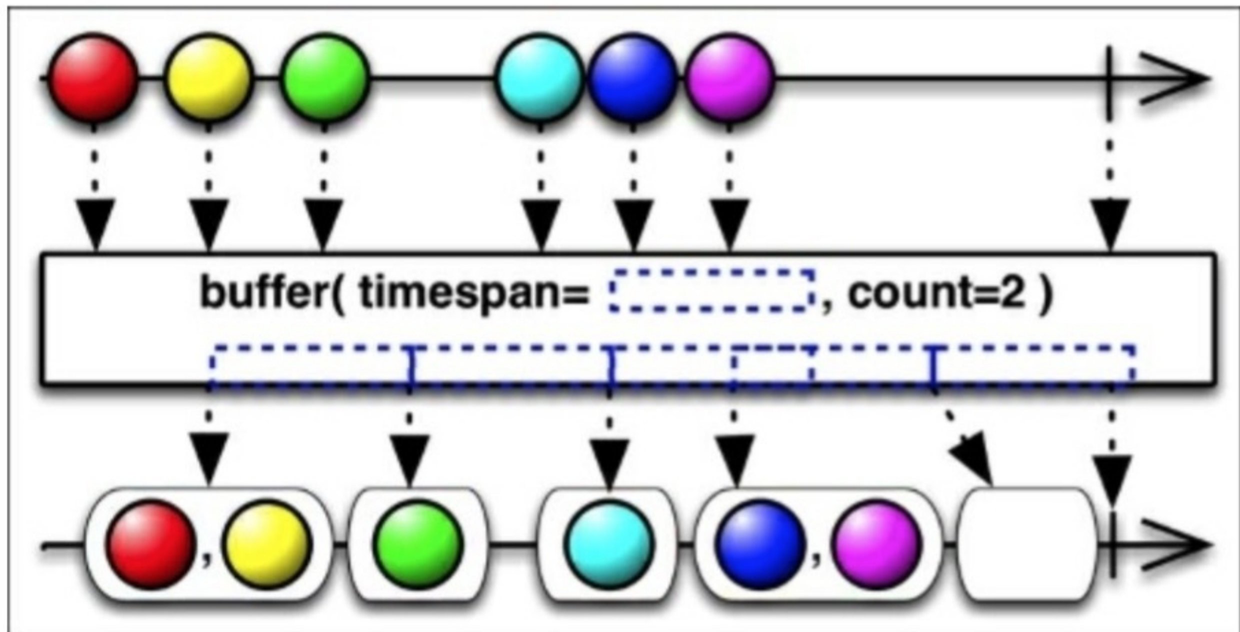
RxJava中的 `buffer()` 函数将源Observable变换一个新的Observable，这个新的Observable每次发射一组列表值而不是一个一个发射。



上图中展示了 `buffer()` 如何将 `count` 作为一个参数来指定有多少数据项被包在发射的列表中。实际上， `buffer()` 函数有几种变体。其中有一个是允许你指定一个 `skip` 值：此后每 `skip` 项数据，然后又用count项数据填充缓冲区。如下图所示：

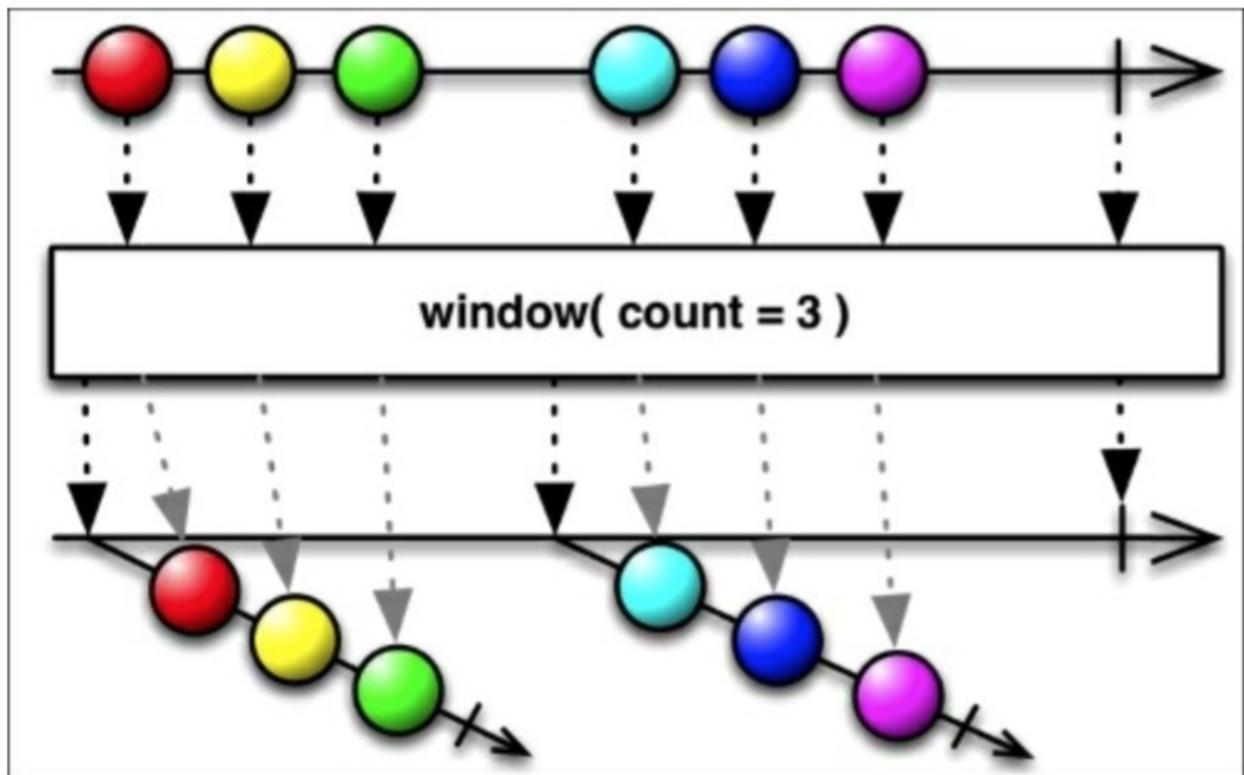


`buffer()` 带一个 `timespan` 的参数，会创建一个每隔timespan时间段就会发射一个列表的Observable。

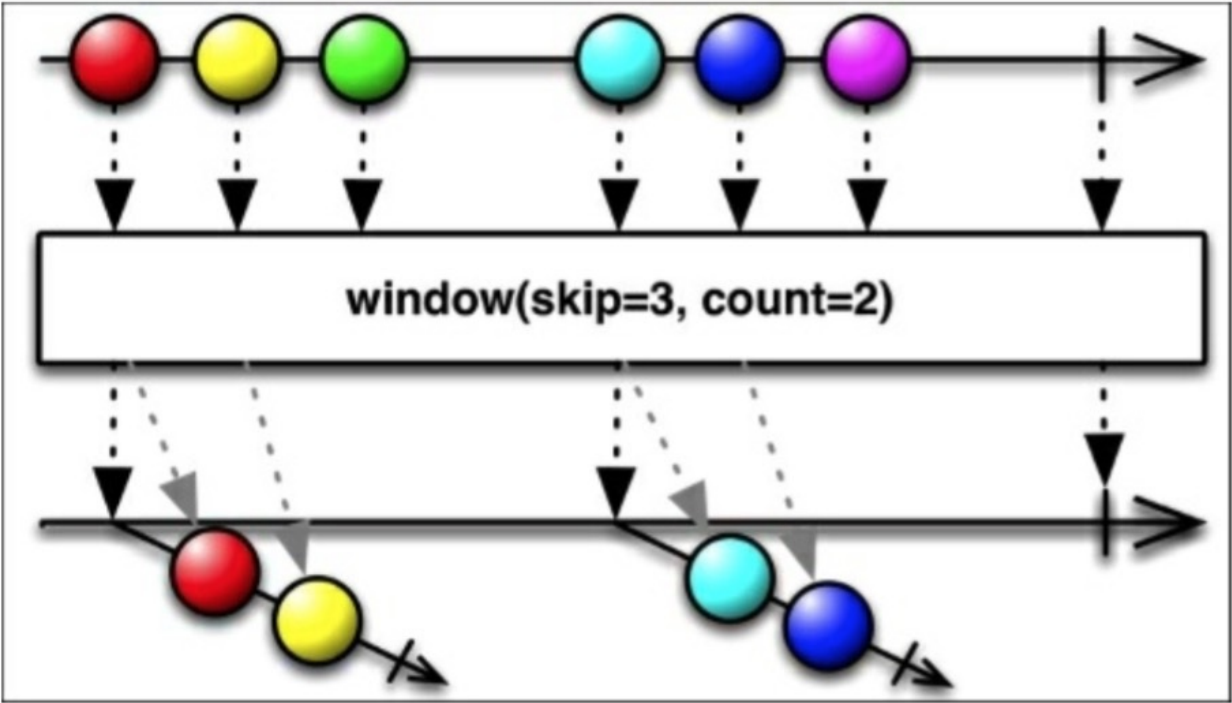


Window

RxJava的 `window()` 函数和 `buffer()` 很像，但是它发射的是Observable而不是列表。下图展示了 `window()` 如何缓存3个数据项并把它们作为一个新的Observable发射出去。

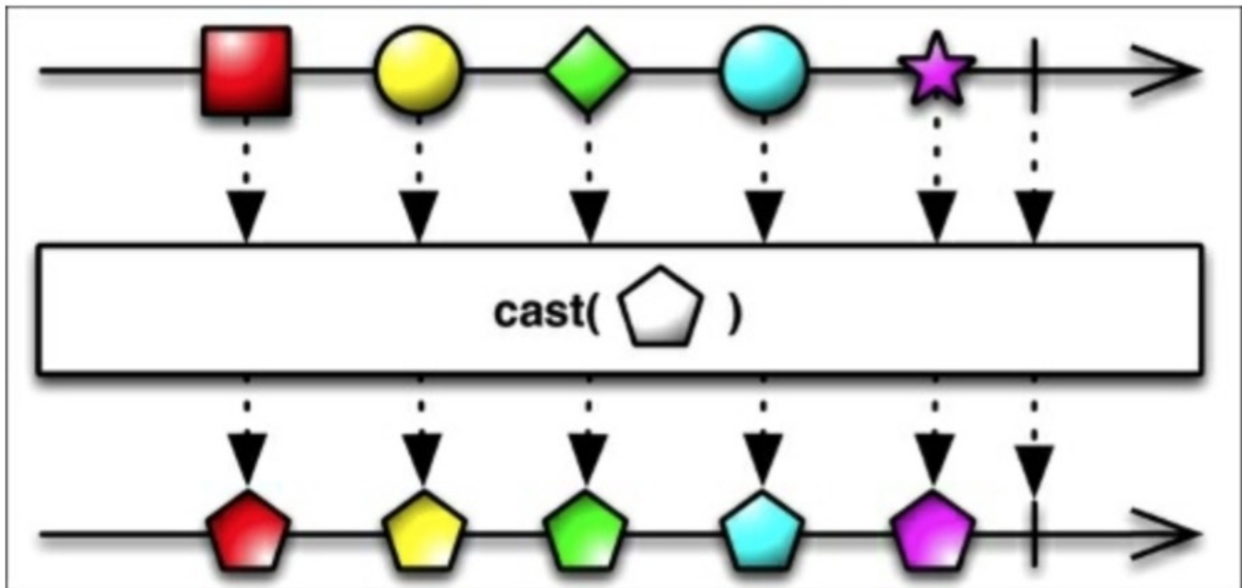


这些Observables中的每一个都发射原始Observable数据的一个子集，数量由 `count` 指定,最后发射一个 `onCompleted()` 结束。正如 `buffer()` 一样, `window()` 也有一个 `skip` 变体,如下图所示：


window(skip=3, count=2)

Cast

RxJava的 `cast()` 函数是本章中最后一个操作符。它是 `map()` 操作符的特殊版本。它将源Observable中的每一项数据都转换为新的类型，把它变成了不同的 `Class` 。



总结

这一章中，我们学习了RxJava时如何控制和转换可观测序列。用我们现在所学的知识，我们可以创建、过滤、转换我们所想要的任何种类的可观测序列。

下一章，我们将学习如何组合Observable，合并它们，连接它们，再或者打包它们。

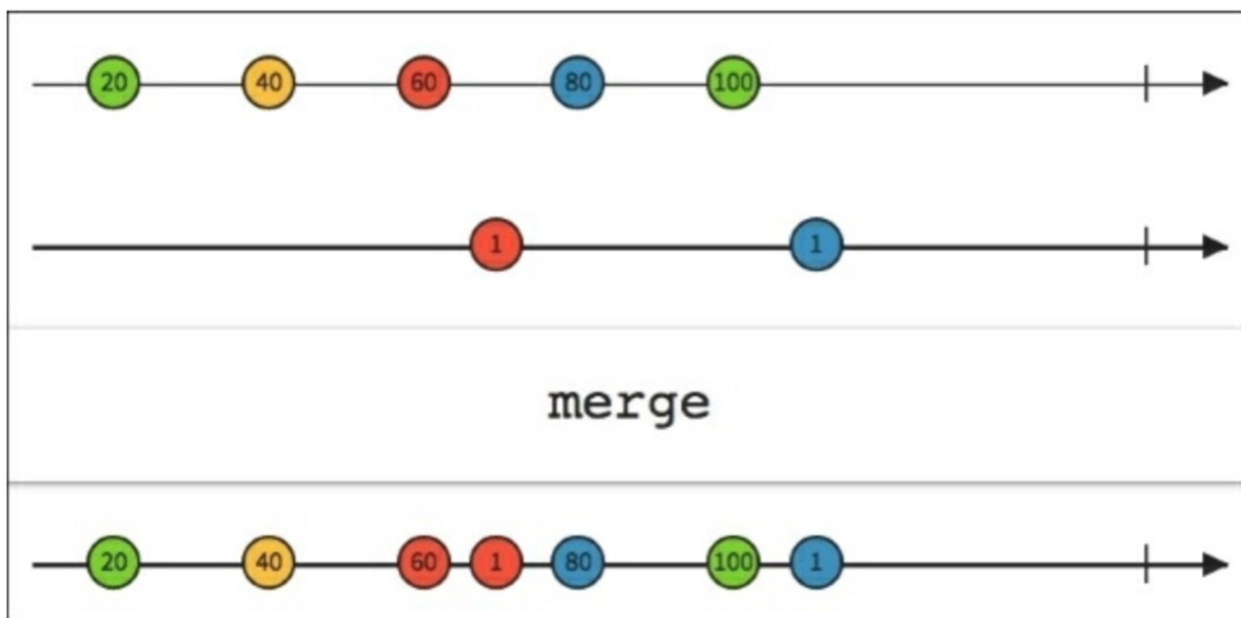
组合Observables

上一章中，我们学到如何转换可观测序列。我们也看到了`map()`,`scan()`,`groupBY()`,以及更多有用的函数的实际例子，它们帮助我们操作Observable来创建我们想要的Observable。

本章中，我们将研究组合函数并学习如何同时处理多个Observables来创建我们想要的Observable。

Merge

在”异步的世界“中经常会创建这样的场景，我们有多个来源但是又只想有一个结果：多输入，单输出。RxJava的 `merge()` 方法将帮助你把两个甚至更多的Observables合并到他们发射的数据项里。下图给出了把两个序列合并在一个最终发射的Observable。



正如你看到的那样，发射的数据被交叉合并到一个Observable里面。注意如果你同步的合并Observable，它们将连接在一起并且不会交叉。

像往常一样，我们用我们的App和已安装的App列表来创建了一个“真实世界”的例子。为此我们还需要第二个Observable。我们可以创建一个单独的应用列表然后让它逆序排列。当然这没有实际的意义，只是为了这个例子。对于第二个列表，我们的 `loadList()` 函数像下面这样：

```
private void loadList(List<AppInfo> apps) {
    mRecyclerView.setVisibility(View.VISIBLE);
    List reversedApps = Lists.reverse(apps);
    Observable<AppInfo> observableApps =Observable.from(apps);
    Observable<AppInfo> observableReversedApps =Observable.from(
reversedApps);
    Observable<AppInfo> mergedObserbable = Observable.merge(obse
rvableApps,observableReversedApps);

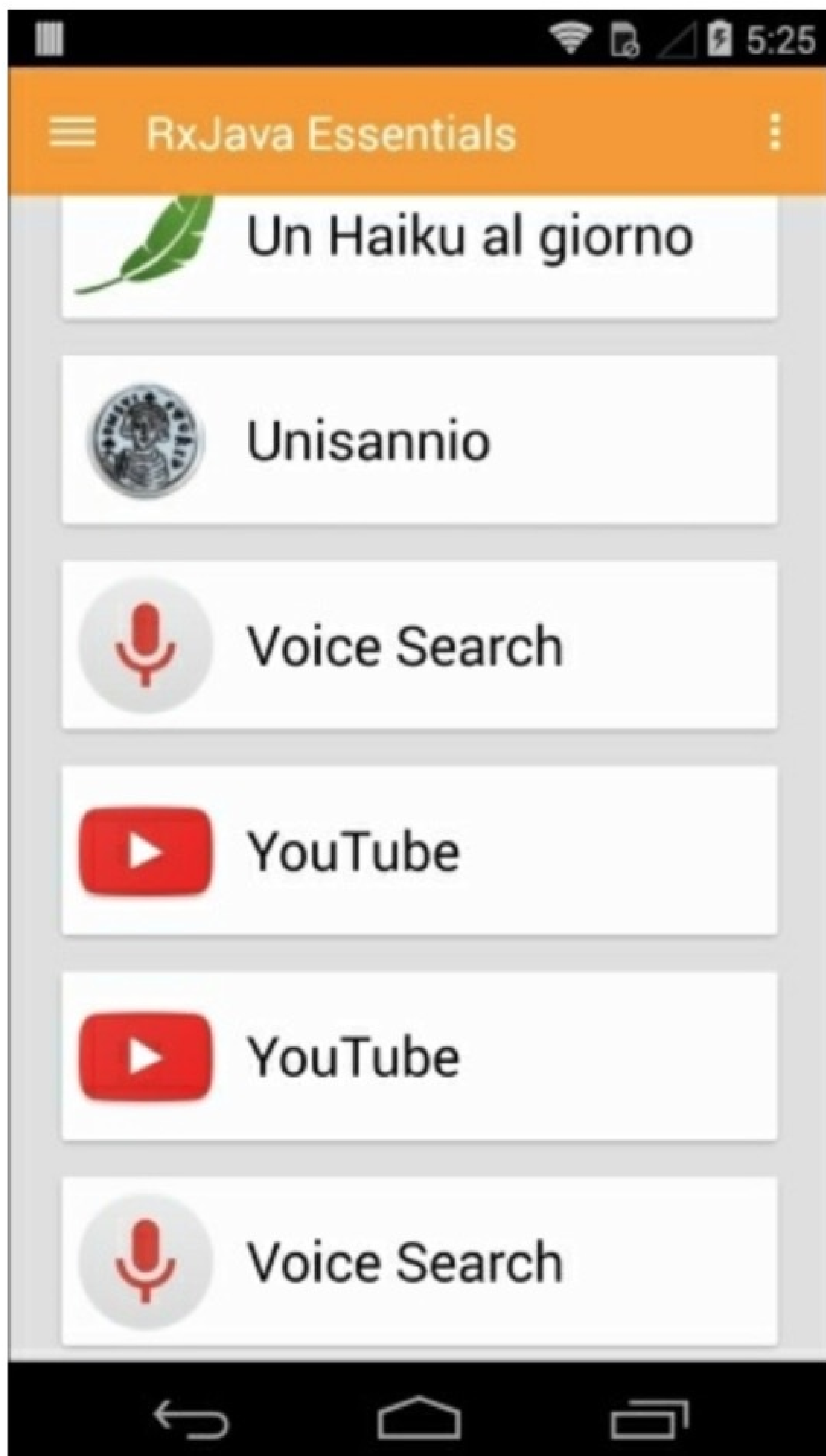
    mergedObserbable.subscribe(new Observer<AppInfo>(){
        @Override
        public void onCompleted() {
            mSwipeRefreshLayout.setRefreshing(false);
            Toast.makeText(getActivity(), "Here is the list!", T
oast.LENGTH_LONG).show();
        }

        @Override
        public void onError(Throwable e) {
            Toast.makeText(getActivity(), "One of the two Observ
able threw an error!", Toast.LENGTH_SHORT).show();
            mSwipeRefreshLayout.setRefreshing(false);
        }

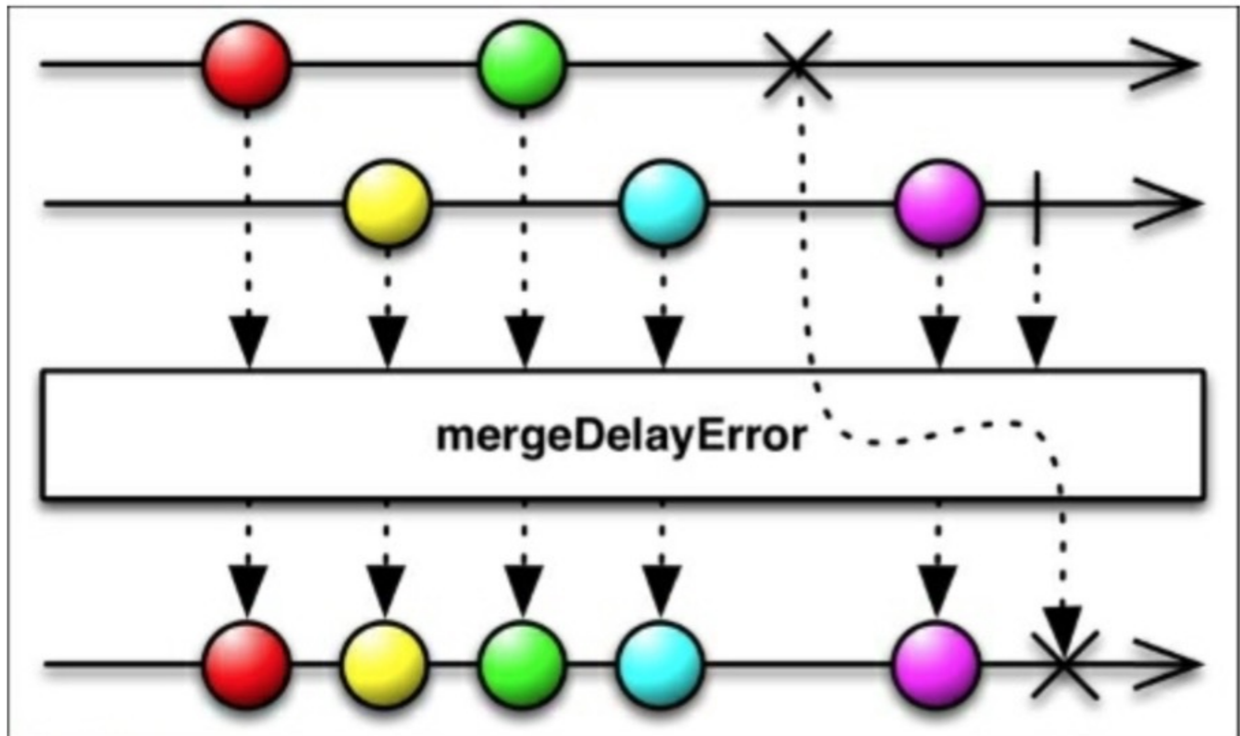
        @Override
        public void onNext(AppInfoappInfo) {
            mAddedApps.add(appInfo);
            mAdapter.addApplication(mAddedApps.size() - 1, appIn
fo);
        }
    });
}
```

我们创建了Observable和observableApps数据项以及新的observableReversedApps逆序列表。使用 `Observable.merge()` ，我们可以创建新的 `Observable MergedObservable` ，它在单个可观测序列中发射源Observables发出的所有数据。

正如你能看到的,每个方法签名都是一样的，因此我们的观察者无需在意任何不同就可以复用代码。结果如下：

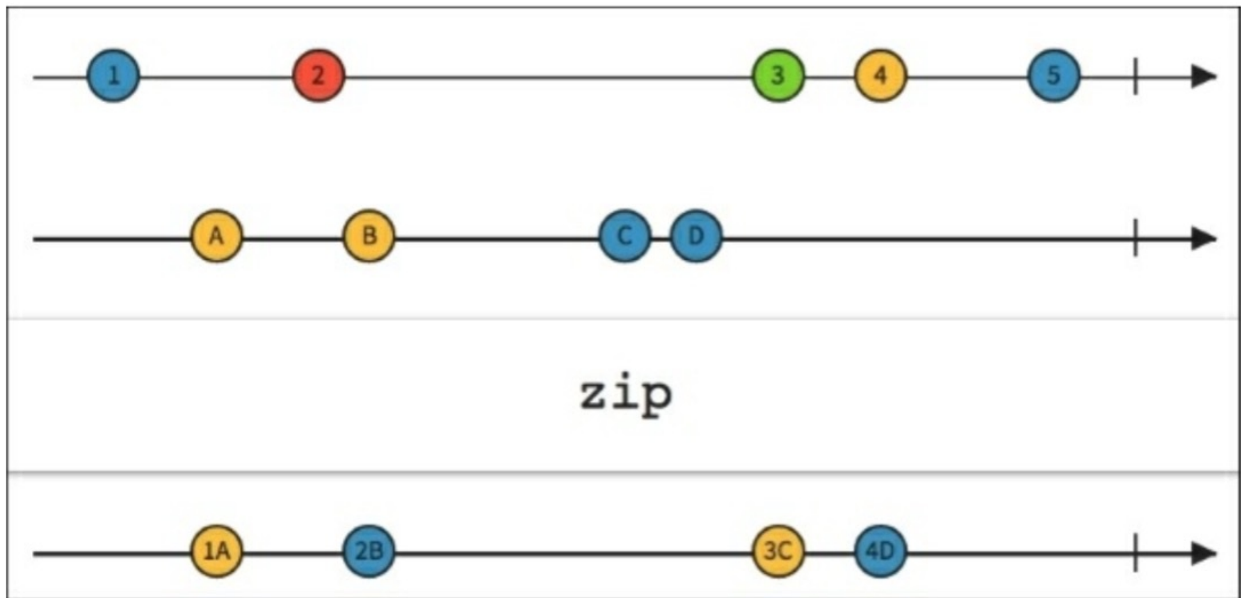


注意错误时的toast消息，你可以认为每个Observable抛出的错误都将会打断合并。如果你需要避免这种情况，RxJava提供了 `mergeDelayError()` ，它能从一个Observable中继续发射数据即便是其中有一个抛出了错误。当所有的Observables都完成时， `mergeDelayError()` 将会发射 `onError()` ，如下图所示：



ZIP

在一种新的可能场景中处理多个数据来源时会带来：多从个Observables接收数据，处理它们，然后将它们合并成一个新的可观测序列来使用。RxJava有一个特殊的方法可以完成：`zip()`合并两个或者多个Observables发射出的数据项，根据指定的函数`Func*`变换它们，并发射一个新值。下图展示了`zip()`方法如何处理发射的“numbers”和“letters”然后将它们合并一个新的数据项：



对于“真实世界”的例子来说，我们将使用已安装的应用列表和一个新的动态的Observable来让例子变得有点有趣味。

```
Observable<Long> tictoc = Observable.interval(1, TimeUnit.SECOND
S);
```

`tictoc` Observable变量使用`interval()`函数每秒生成一个Long类型的数据：虽简单但有效，正如之前所说的，我们需要一个`Func`对象。因为它需要传两个参数，所以是`Func2`：

```
private AppInfo updateTitle(AppInfoappInfo, Long time) {
    appInfo.setName(time + " " + appInfo.getName());
    return appInfo;
}
```

现在我们的 `loadList()` 函数变成这样：

```
private void loadList(List<AppInfo> apps) {
    mRecyclerView.setVisibility(View.VISIBLE);
    Observable<AppInfo> observableApp = Observable.from(apps);

    Observable<Long> tictoc = Observable.interval(1, TimeUnit.SE
CONDS);

    Observable.zip(observableApp, tictoc,
    (AppInfo appInfo, Long time) -> updateTitle(appInfo, time))
    .observeOn(AndroidSchedulers.mainThread())
    .subscribe(new Observer<AppInfo>() {
        @Override
        public void onCompleted() {
            Toast.makeText(getActivity(), "Here is the list!", T
oast.LENGTH_LONG).show();
        }

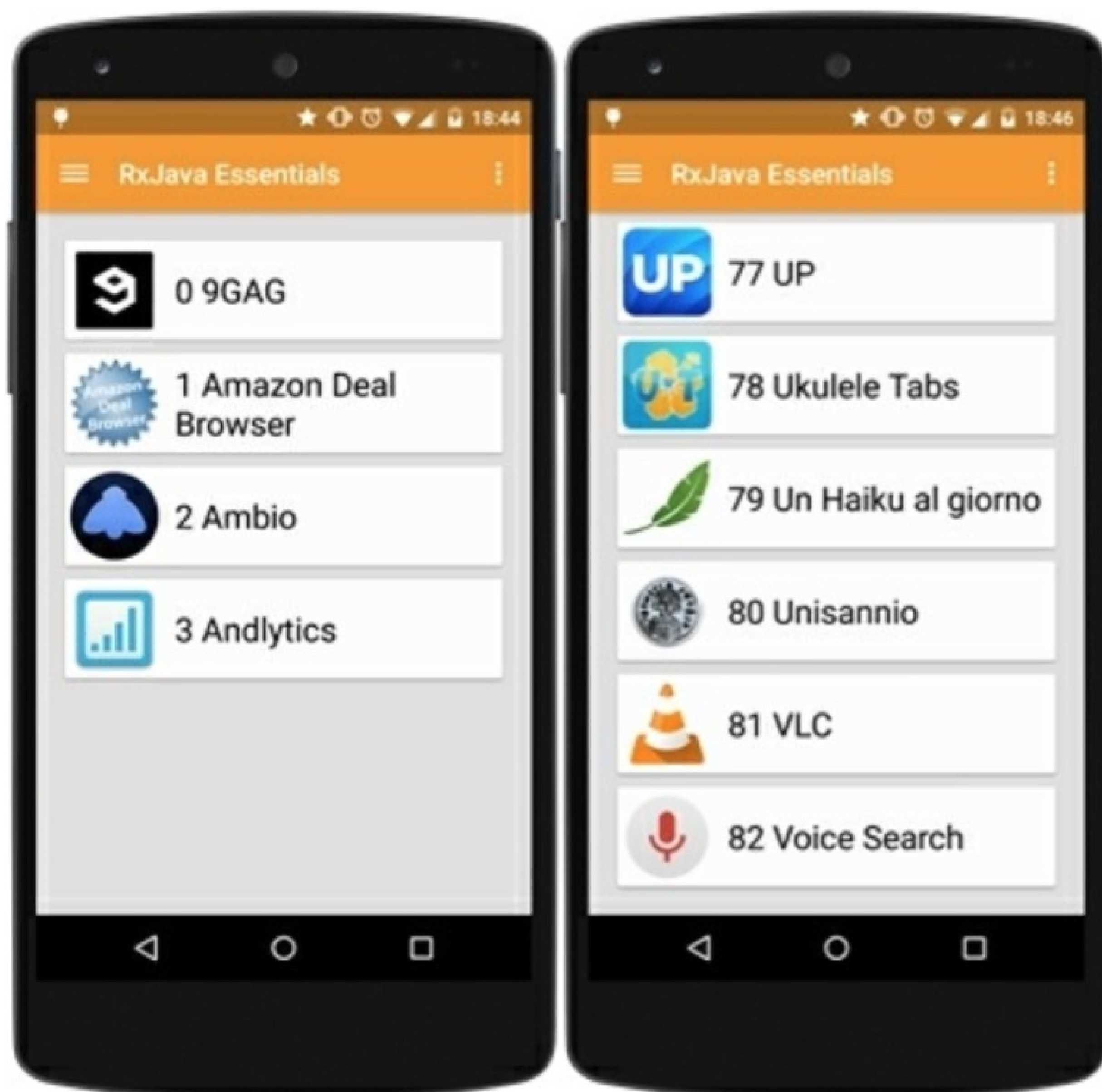
        @Override
        public void onError(Throwable e) {
            mSwipeRefreshLayout.setRefreshing(false);
            Toast.makeText(getActivity(), "Something went wrong!"
, Toast.LENGTH_SHORT).show();
        }

        @Override
        public void onNext(AppInfoappInfo) {
            if (mSwipeRefreshLayout.isRefreshing()) {
                mSwipeRefreshLayout.setRefreshing(false);
            }
            mAddedApps.add(appInfo);
            int position = mAddedApps.size() - 1;
            mAdapter.addApplication(position, appInfo);
            mRecyclerView.smoothScrollToPosition(position);
        }
    });
}
```

正如你看到的那样，`zip()` 函数有三个参数：两个Observables和一个 `Func2` 。

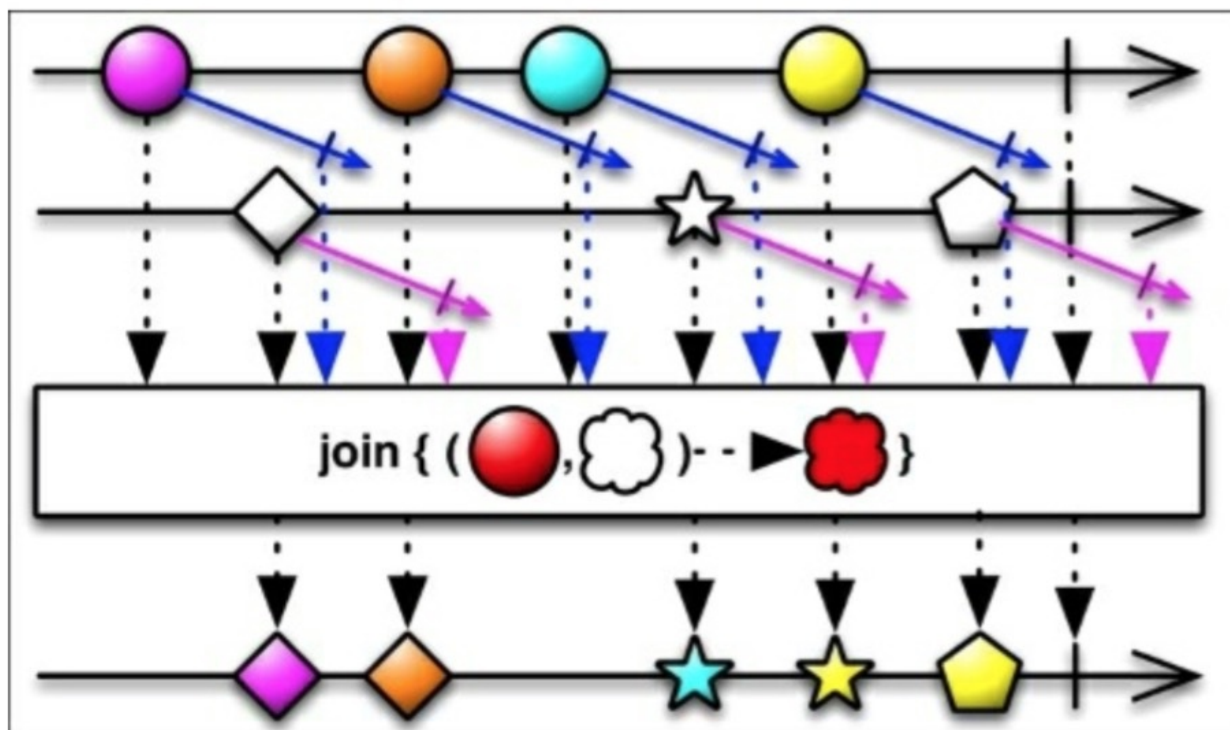
仔细一看会发现 `observeOn()` 函数。它将在下一章中讲解：现在我们可以小试一下。

结果如下：



Join

前面两个方法，`zip()` 和 `merge()` 方法作用在发射数据的范畴内，在决定如何操作值之前有些场景我们需要考虑时间的。RxJava的 `join()` 函数基于时间窗口将两个Observables发射的数据结合在一起。



为了正确的理解上一张图，我们解释下 `join()` 需要的参数：

- 第二个Observable和源Observable结合。
- `Func1` 参数：在指定的由时间窗口定义时间间隔内，源Observable发射的数据和从第二个Observable发射的数据相互配合返回的Observable。
- `Func1` 参数：在指定的由时间窗口定义时间间隔内，第二个Observable发射的数据和从源Observable发射的数据相互配合返回的Observable。
- `Func2` 参数：定义已发射的数据如何与新发射的数据项相结合。
- 如下练习的例子，我们可以修改 `loadList()` 函数像下面这样：

```
private void loadList(List<AppInfo> apps) {
    mRecyclerView.setVisibility(View.VISIBLE);

    Observable<AppInfo> appsSequence =
    Observable.interval(1000, TimeUnit.MILLISECONDS)
```

```
                .map(position -> {
                    return apps.get(position.intValue());
                });

    Observable<Long> tictoc = Observable.interval(1000,TimeUni
t.MILLISECONDS);

    appsSequence.join(
        tictoc,
        appInfo -> Observable.timer(2,TimeUnit.SECONDS),
        time -> Observable.timer(0, TimeUnit.SECONDS),
        this::updateTitle)
        .observeOn(AndroidSchedulers.mainThread())
        .take(10)
        .subscribe(new Observer<AppInfo>() {
            @Override
            public void onCompleted() {
                Toast.makeText(getActivity(), "Here is the lis
t!", Toast.LENGTH_LONG).show();
            }

            @Override
            public void onError(Throwable e) {
                mSwipeRefreshLayout.setRefreshing(false);
                Toast.makeText(getActivity(), "Something went
wrong!", Toast.LENGTH_SHORT).show();
            }

            @Override
            public void onNext(AppInfoappInfo) {
                if (mSwipeRefreshLayout.isRefreshing()) {
                    mSwipeRefreshLayout.setRefreshing(false);
                }
                mAddedApps.add(appInfo);
                int position = mAddedApps.size() - 1;
                mAdapter.addApplication(position, appInfo);
                mRecyclerView.smoothScrollToPosition(position)
;
            }
        });
```

```
}
```

我们有一个新的对象 `appsSequence` ，它是一个每秒从我们已安装的app列表发射app数据的可观测序列。 `tictoc` 这个Observable数据每秒只发射一个新的 `Long` 型整数。为了合并它们，我们需要指定两个 `Func1` 变量：

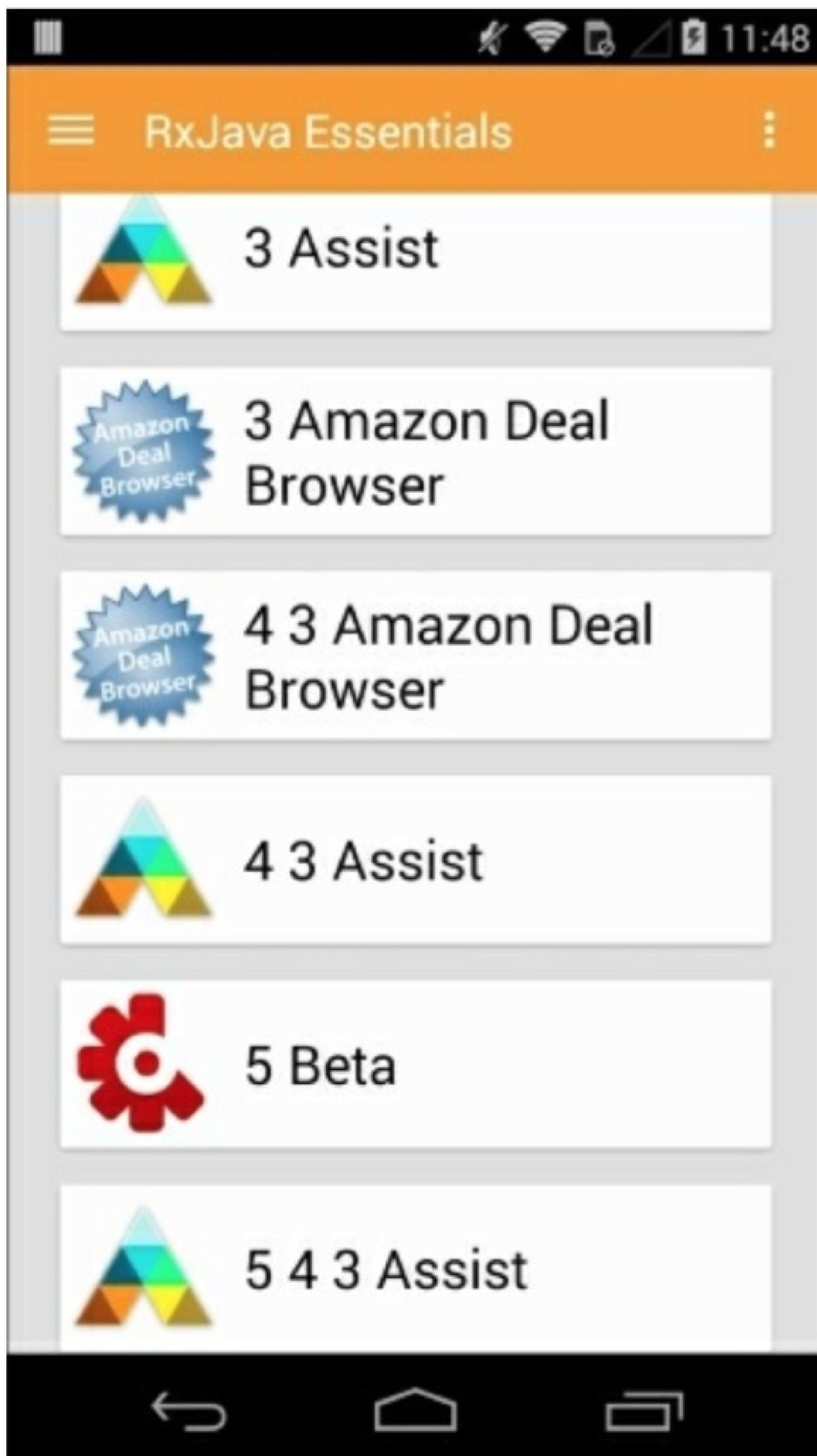
```
appInfo -> Observable.timer(2, TimeUnit.SECONDS)

time -> Observable.timer(0, TimeUnit.SECONDS)
```

上面描述了两个时间窗口。下面一行描述我们如何使用 `Func2` 将两个发射的数据结合在一起。

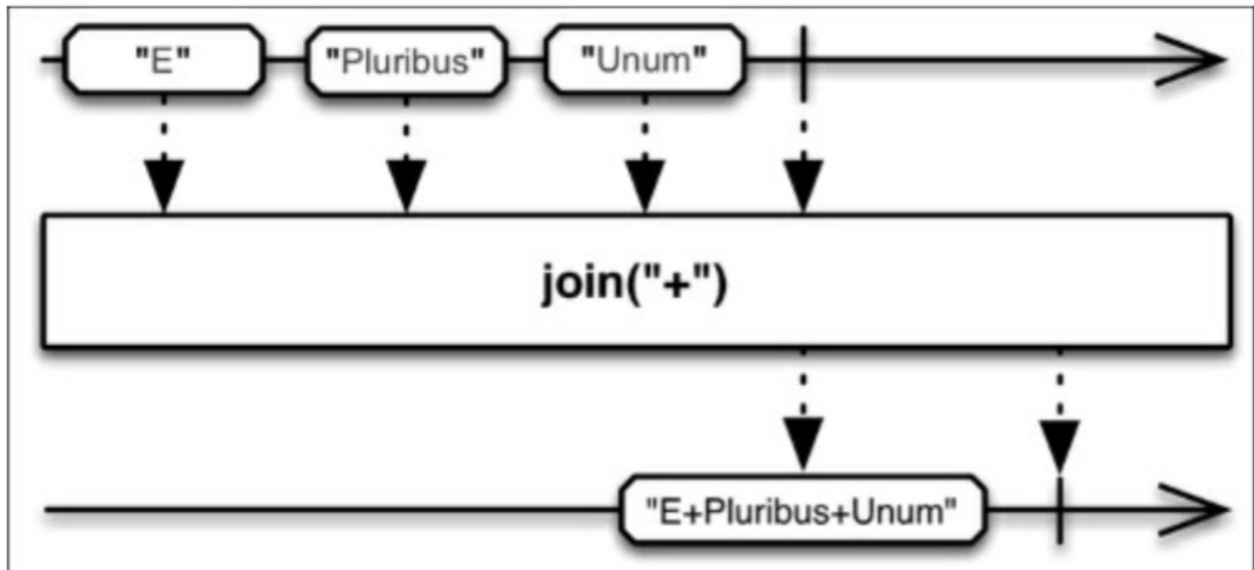
```
this::updateTitle
```

结果如下：


11:48
RxJava Essentials
3 Assist
3 Amazon Deal Browser
4 3 Amazon Deal Browser
4 3 Assist
5 Beta
5 4 3 Assist

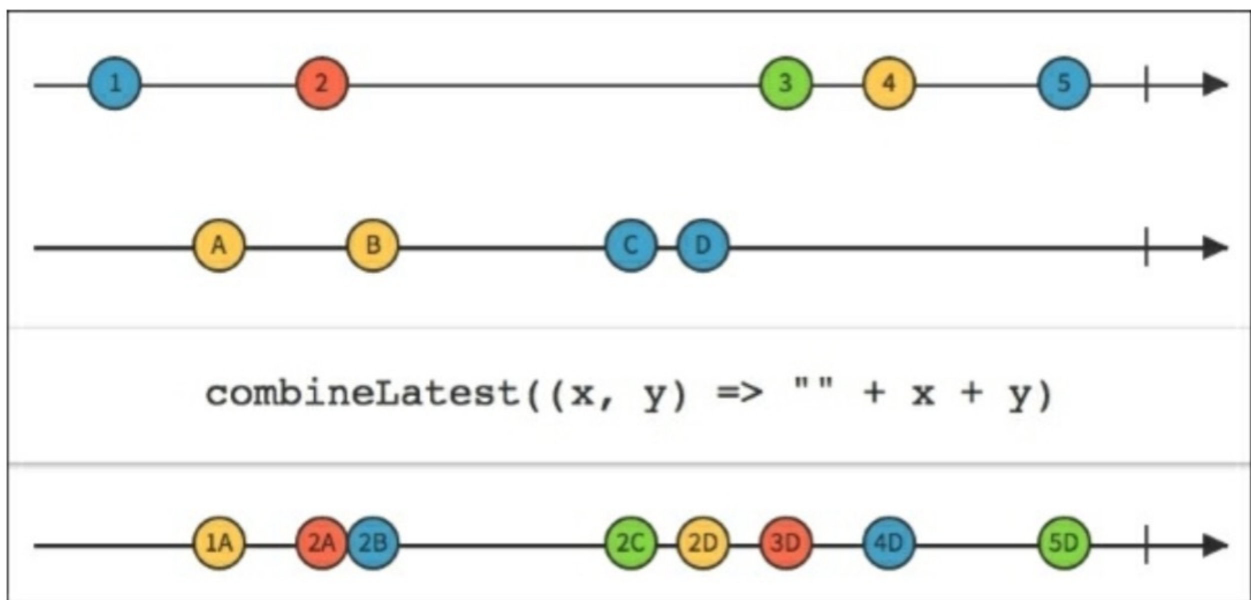
它看起来有点乱，但是注意app的名字和我们指定的时间窗口，我们可以看到：一旦第二个数据发射了我们就会将它与源数据结合，但我们用同一个源数据有2秒钟。这就是为什么标题重复数字增加的原因。

值得一提的是，为了简单起见，也有一个 `join()` 操作符作用于字符串然后简单的和发射的字符串连接成最终的字符串。



combineLatest

RxJava的 `combineLatest()` 函数有点像 `zip()` 函数的特殊形式。正如我们已经学习的， `zip()` 作用于最近未打包的两个Observables。相反， `combineLatest()` 作用于最近发射的数据项：如果 `Observable1` 发射了A并且 `Observable2` 发射了B和C， `combineLatest()` 将会分组处理AB和AC，如下图所示：



`combineLatest()` 函数接受二到九个Observable作为参数，如果有需要的话或者单个Observables列表作为参数。

从之前的例子中把 `loadList()` 函数借用过来，我们可以修改一下来用于 `combineLatest()` 实现“真实世界”这个例子：

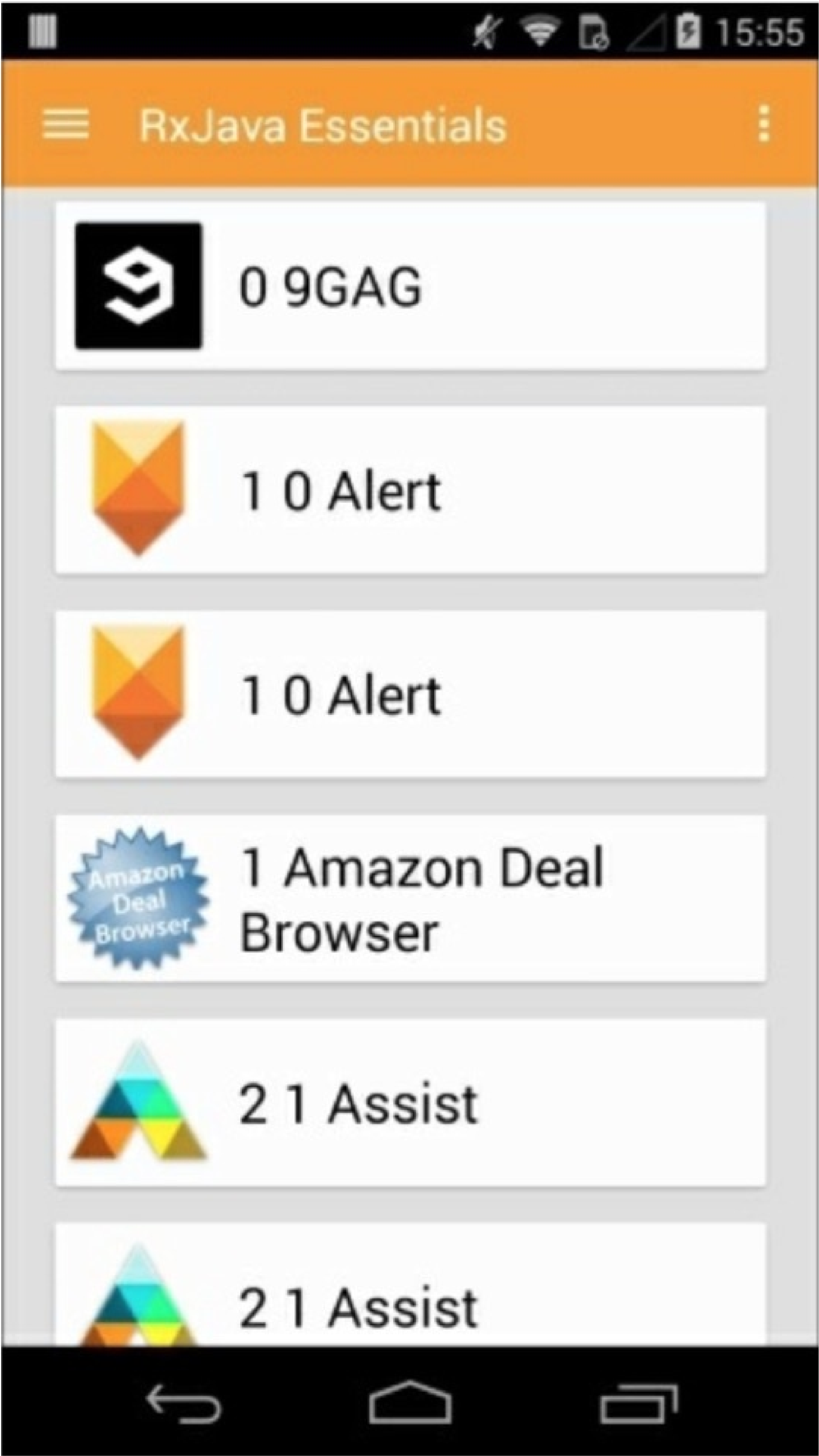
```
private void loadList(List<AppInfo> apps) {
    mRecyclerView.setVisibility(View.VISIBLE);
    Observable<AppInfo> appsSequence = Observable.interval(1000,
 TimeUnit.MILLISECONDS)
                .map(position ->apps.get(position.intValue()));
    Observable<Long> tictoc = Observable.interval(1500, TimeUnit
.MILLISECONDS);
    Observable.combineLatest(appsSequence, tictoc,
                this::updateTitle)
       .observeOn(AndroidSchedulers.mainThread())
        .subscribe(new Observer<AppInfo>() {

        @Override
        public void onCompleted() {
            Toast.makeText(getActivity(), "Here is the list!", T
oast.LENGTH_LONG).show();
        }

        @Override
        public void onError(Throwable e) {
            mSwipeRefreshLayout.setRefreshing(false);
            Toast.makeText(getActivity(), "Something went wrong!"
, Toast.LENGTH_SHORT).show();
        }

        @Override
        public void onNext(AppInfoappInfo) {
            if (mSwipeRefreshLayout.isRefreshing()) {
                mSwipeRefreshLayout.setRefreshing(false);
            }
            mAddedApps.add(appInfo);
            int position = mAddedApps.size() - 1;
            mAdapter.addApplication(position, appInfo);
            mRecyclerView.smoothScrollToPosition(position);
        }
    });
}
```

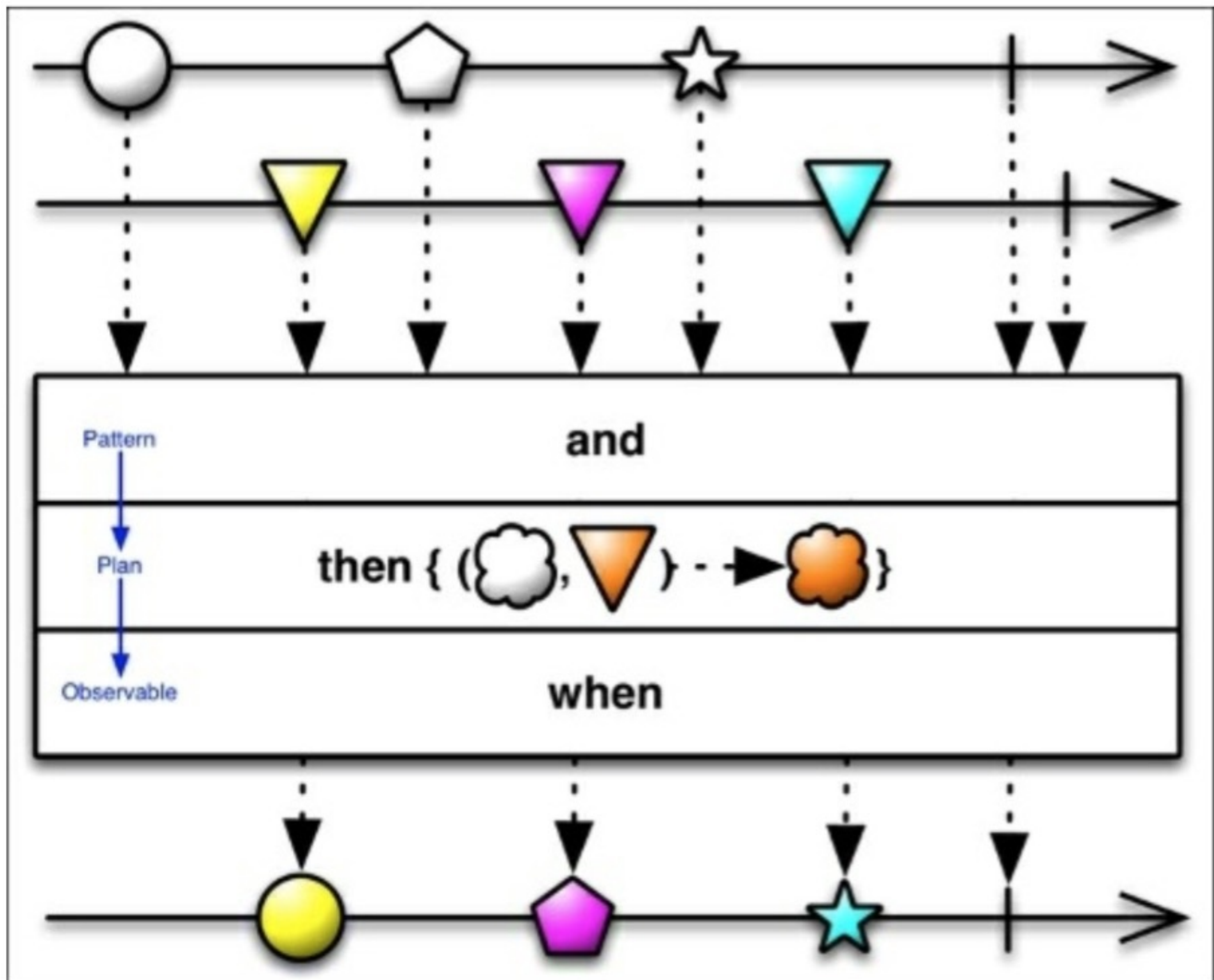
这我们使用了两个Observables：一个是每秒钟从我们已安装的应用列表发射一个App数据，第二个是每隔1.5秒发射一个 `Long` 型整数。我们将他们结合起来并执行 `updateTitle()` 函数，结果如下：


15:55
RxJava Essentials
0 9GAG
1 0 Alert
1 0 Alert
Amazon Deal Browser
1 Amazon Deal Browser
2 1 Assist
2 1 Assist

正如你看到的，由于不同的时间间隔，`AppInfo`对象如我们所预料的那样有时候会重复。

And,Then和When

在将来还有一些 `zip()` 满足不了的场景。如复杂的架构，或者是仅仅为了个人爱好，你可以使用And/Then/When解决方案。它们在RxJava的joins包下，使用Pattern和Plan作为中介，将发射的数据集合并到一起。



我们的 `loadList()` 函数将会被修改从这样：

```
private void loadList(List<AppInfo> apps) {

    mRecyclerView.setVisibility(View.VISIBLE);

    Observable<AppInfo> observableApp = Observable.from(apps);

    Observable<Long> tictoc = Observable.interval(1, TimeUnit.SE
CONDS);
```

```
    Pattern2<AppInfo, Long> pattern = JoinObservable.from(observ
ableApp).and(tictoc);

    Plan0<AppInfo> plan = pattern.then(this::updateTitle);

    JoinObservable
        .when(plan)
        .toObservable()
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe(new Observer<AppInfo>() {

            @Override
            public void onCompleted() {
                Toast.makeText(getActivity(), "Here is the list!"
, Toast.LENGTH_LONG).show();
            }

            @Override
            public void onError(Throwable e) {
                mSwipeRefreshLayout.setRefreshing(false);
                Toast.makeText(getActivity(), "Something went wr
ong!", Toast.LENGTH_SHORT).show();
            }

            @Override
            public void onNext(AppInfoappInfo) {
                if (mSwipeRefreshLayout.isRefreshing()) {
                mSwipeRefreshLayout.setRefreshing(false);
                }
                mAddedApps.add(appInfo);
                int position = mAddedApps.size() - 1;
                mAdapter.addApplication(position, appInfo); mRec
yclerView.smoothScrollToPosition(position);
            }
        });
}
```

和通常一样，我们有两个发射的序列， `observableApp` ，发射我们安装的应用列表数据， `tictoc` 每秒发射一个 `Long` 型整数。现在我们用 `and()` 连接源Observable和第二个Observable。

```
JoinObservable.from(observableApp).and(tictoc);
```

这里创建一个 `pattern` 对象，使用这个对象我们可以创建一个 `Plan` 对象:"我们有两个发射数据的Observables, `then()` 是做什么的？"

```
pattern.then(this::updateTitle);
```

现在我们有了一个 `Plan` 对象并且当plan发生时我们可以决定接下来发生的事情。

```
.when(plan).toObservable()
```

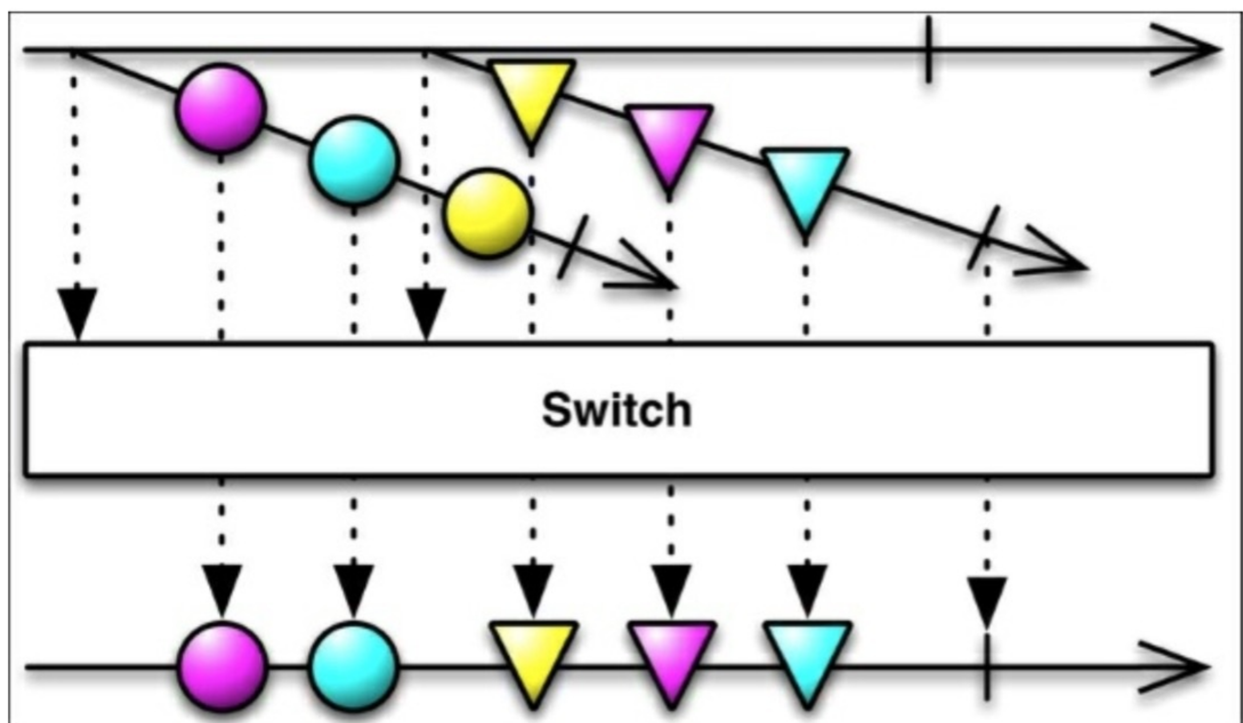
这时候，我们可以订阅新的Observable，正如我们总是做的那样。

Switch

有这样一个复杂的场景就是在一个 `subscribe-unsubscribe` 的序列里我们能够从一个Observable自动取消订阅来订阅一个新的Observable。

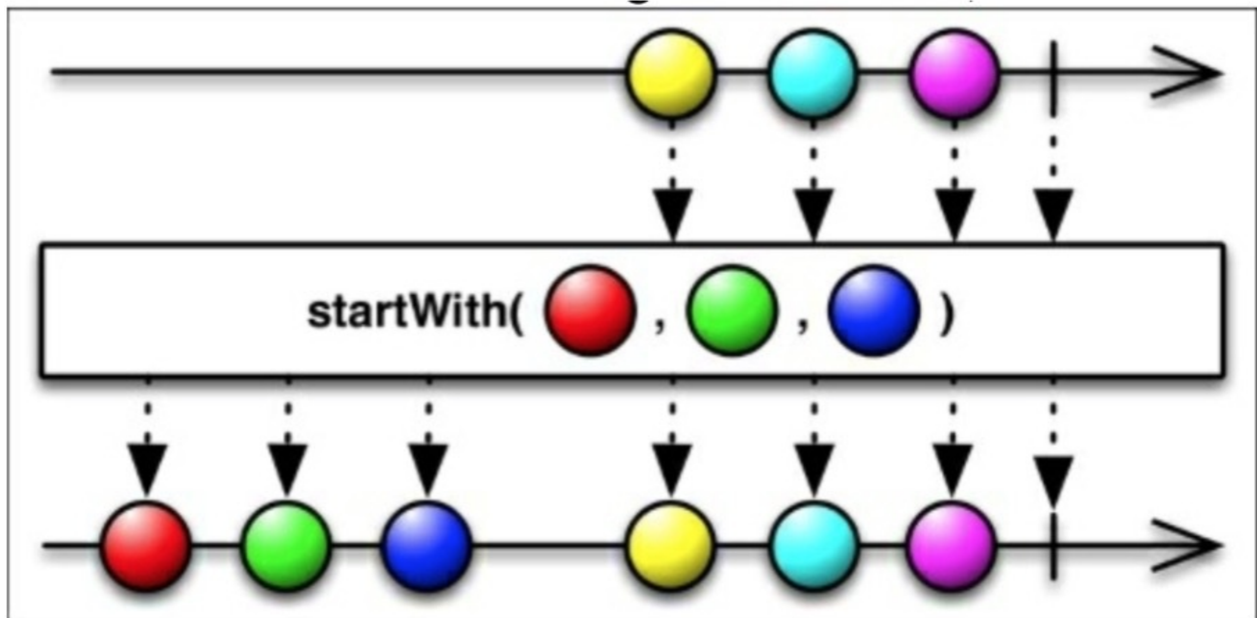
RxJava的 `switch()` ，正如定义的，将一个发射多个Observables的Observable转换成另一个单独的Observable，后者发射那些Observables最近发射的数据项。

给出一个发射多个Observables序列的源Observable， `switch()` 订阅到源Observable然后开始发射由第一个发射的Observable发射的一样的数据。当源Observable发射一个新的Observable时， `switch()` 立即取消订阅前一个发射数据的Observable（因此打断了从它那里发射的数据流）然后订阅一个新的Observable，并开始发射它的数据。



StartWith

我们已经学到如何连接多个Observables并追加指定的值到一个发射序列里。RxJava的 `startWith()` 是 `concat()` 的对应部分。正如 `concat()` 向发射数据的Observable追加数据那样，在Observable开始发射他们的数据之前，`startWith()` 通过传递一个参数来先发射一个数据序列。



总结

这章中，我们学习了如何将两个或者更多个Observable结合来创建一个新的可观测序列。我们将能够 `merge` Observable， `join` Observables ， `zip` Observables 并在几种情况下把他们结合在一起。

下一章，我们将介绍调度器，它将很容易的帮助我们创建主线程以及提高我们应用程序的性能。我们也将学习如何正确的执行长任务或者I/O任务来获得更好的性能。

Schedulers-解决Android主线程问题

前面一章是最后一章关于RxJava的Observable的创建和操作的章节。我们学习到了如何将两个或更多的Observables合并在一起，`join`它们，`zip`它们，`merge`它们以及如何创建一个新的Observable来满足我们特殊的需求。

本章中，我们提升标准看看如何使用RxJava的调度器来处理多线程和并发编程的问题。我们将学习到如何以响应式的方式创建网络操作，内存访问，以及耗时任务。

StrictMode

为了获得更多出现在代码中的关于公共问题的信息，我们激活了 `StrictMode` 模式。

`StrictMode` 帮助我们侦测敏感的活动，如我们无意的在主线程执行磁盘访问或者网络调用。正如你所知道的，在主线程执行繁重的或者长时的任务是不可取的。因为Android应用的主线程时UI线程，它被用来处理和UI相关的操作：这也是获得更平滑的动画体验和响应式App的唯一方法。

为了在我们的App中激活 `StrictMode` ，我们只需要在 `MainActivity` 中添加几行代码，即 `onCreate()` 方法中这样：

```
@Override
public void onCreate() {
    super.onCreate();
    if (BuildConfig.DEBUG) {
        StrictMode.setThreadPolicy(new StrictMode.ThreadPolicy.Builder().detectAll().penaltyLog().build());
        StrictMode.setVmPolicy(new StrictMode.VmPolicy.Builder().detectAll().penaltyLog().build());
    }
}
```

我们并不想它总是激活着，因此我们只在debug构建时使用。这种配置将报告每一种关于主线程用法的违规做法，并且这些做法都可能与内存泄露有关： `Activities` 、 `BroadcastReceivers` 、 `Sqlite` 等对象。

选择了 `penaltyLog()` ，当违规做法发生时， `StrictMode` 将会在logcat打印一条信息。

避免阻塞I/O的操作

阻塞I/O的操作会导致App必须等待结果返回（阻塞结束）才能进行下一步操作。在UI线程上执行一个阻塞操作会将UI强行卡住，直接造成很糟糕的用户体验。

我们激活 `StrictMode` 后，我们开始收到了关于我们的App错误操作磁盘I/O的不良信息。

```
D/StrictMode  StrictMode policy violation; ~duration=998 ms: android.os.StrictMode$StrictModeDiskReadViolation: policy=31 violation=2
at android.os.StrictMode$AndroidBlockGuardPolicy.onReadFromDisk(StrictMode.java:1135)
at libcore.io.BlockGuardOs.open(BlockGuardOs.java:106) at libcore.io.IoBridge.open(IoBridge.java:393)
at java.io.FileOutputStream.<init>(FileOutputStream.java:88)
at android.app.ContextImpl.openFileOutput(ContextImpl.java:918)
at android.content.ContextWrapper.openFileOutput(ContextWrapper.java:185)
at com.packtpub.apps.rxjava_essentials.Utils.storeBitmap (Utils.java:30)
```

上一条信息告诉我们 `Utils.storeBitmap()` 函数执行完耗时998ms：在UI线程上近1秒的不必要的工作和App上近1秒不必要的迟钝。这是因为我们以阻塞的方式访问磁盘。我们的 `storeBitmap()` 函数包含了：

```
FileOutputStream fOut = context.openFileOutput(filename, Context.MODE_PRIVATE);
```

它直接访问智能手机的固态存储然后就慢了。我们该如何提高访问速度呢？ `storeBitmap()` 函数保存了已安装App的图标。他的返回值类型为 `void` ，因此在执行下一个操作前我们毫无理由去等待直到它完成。我们可以启动它并让它执行在不同的线程。近几年来Android的线程管理发生了许多变化，导致App出现诡

异的行为。我们可以使用 `AsyncTask` ，但是我们要避免掉入前几章里的 `onPre... onPost...doInBackGround` 地狱。下面我们将换用RxJava的方式。调度器万岁！

Schedulers

调度器以一种最简单的方式将多线程用在你的Apps的中。它们时RxJava重要的一部分并能很好地与Observables协同工作。它们无需处理实现、同步、线程、平台限制、平台变化而可以提供一种灵活的方式来创建并发程序。

RxJava提供了5种调度器：

- `.io()`
- `.computation()`
- `.immediate()`
- `.newThread()`
- `.trampoline()`

让我们一个一个的来看下它们：

Schedulers.io()

这个调度器时用于I/O操作。它基于根据需要，增长或缩减来自适应的线程池。我们将使用它来修复我们之前看到的 `StrictMode` 违规做法。由于它专用于I/O操作，所以并不是RxJava的默认方法；正确的使用它是由开发者决定的。

重点需要注意的是线程池是无限制的，大量的I/O调度操作将创建许多个线程并占用内存。一如既往的是，我们需要在性能和简捷两者之间找到一个有效的平衡点。

Schedulers.computation()

这个是计算工作默认的调度器，它与I/O操作无关。它也是许多RxJava方法的默认调度
器： `buffer()` , `debounce()` , `delay()` , `interval()` , `sample()` , `skip()`
。

Schedulers.immediate()

这个调度器允许你立即在当前线程执行你指定的工作。它是 `timeout()` , `timeInterval()` ,以及 `timestamp()` 方法默认的调度器。

Schedulers.newThread()

这个调度器正如它所看起来的那样：它为指定任务启动一个新的线程。

Schedulers.trampoline()

当我们想在当前线程执行一个任务时，并不是立即，我们可以用 `.trampoline()` 将它入队。这个调度器将会处理它的队列并且按序运行队列中每一个任务。它是 `repeat()` 和 `retry()` 方法默认的调度器。

非阻塞I/O操作

现在我们知道如何在一个指定I/O调度器上来调度一个任务，我们可以修改 `storeBitmap()` 函数并再次检查 `StrictMode` 的不合规做法。为了这个例子，我们可以在新的 `blockingStoreBitmap()` 函数中重排代码。

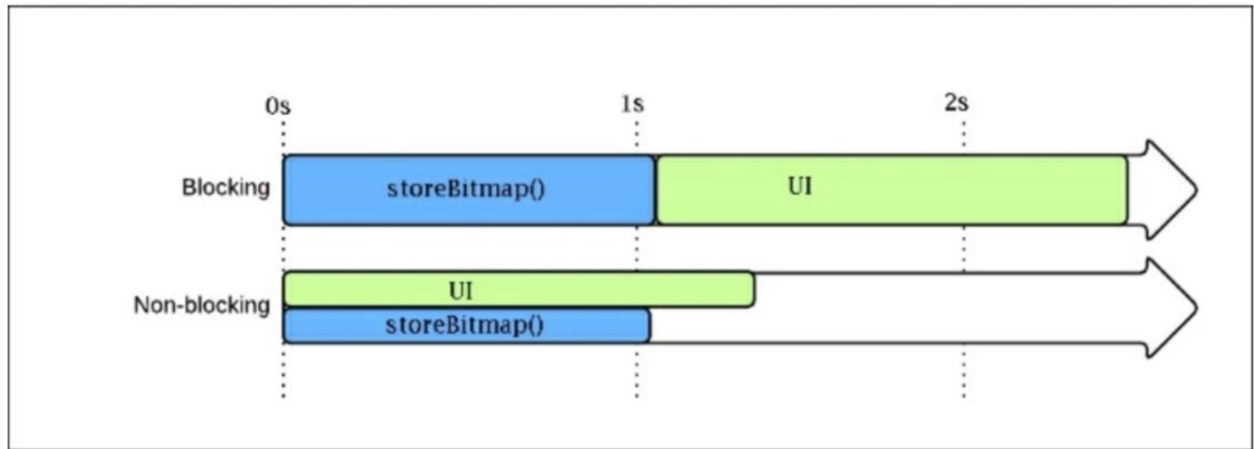
```
private static void blockingStoreBitmap(Context context, Bitmap
bitmap, String filename) {
    FileOutputStream fOut = null;
    try {
        fOut = context.openFileOutput(filename, Context.MODE_PRI
VATE);
        bitmap.compress(Bitmap.CompressFormat.PNG, 100, fOut);
        fOut.flush();
        fOut.close();
    } catch (Exception e) {
        throw new RuntimeException(e);
    } finally {
        try {
            if (fOut != null) {
                fOut.close();
            }
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }
}
```

现在我们可以使用 `Schedulers.io()` 创建非阻塞的版本：

```
public static void storeBitmap(Context context, Bitmap bitmap, S
tring filename) {
    Schedulers.io().createWorker().schedule(() -> {
        blockingStoreBitmap(context, bitmap, filename);
    });
}
```

每次我们调用`storeBitmap()`，RxJava处理创建所有它需要从I / O线程池一个特定的I/ O线程执行我们的任务。所有要执行的操作都避免在UI线程执行并且我们的App比之前要快上1秒：logcat上也不再有`StrictMode`的不合规做法。

下图展示了我们在`storeBitmap()`场景看到的两种方法的不同：



SubscribeOn and ObserveOn

我们学到了如何在一个调度器上运行一个任务。但是我们如何利用它来和Observables一起工作呢？RxJava提供了 `subscribeOn()` 方法来用于每个Observable对象。 `subscribeOn()` 方法用 `Scheduler` 来作为参数并在这个Scheduler上执行Observable调用。

在“真实世界”这个例子中，我们调整 `loadList()` 函数。首先，我们需要一个新的 `getApps()` 方法来检索已安装的应用列表：

```
private Observable<AppInfo> getApps() {
    return Observable.create(subscriber -> {
        List<AppInfo> apps = new ArrayList<>();
        SharedPreferences sharedPref = getActivity().getPreferen
ces(Context.MODE_PRIVATE);
        Type appInfoType = new TypeToken<List<AppInfo>>(){}.getT
ype();
        String serializedApps = sharedPref.getString("APPS", "")
;
        if (!"".equals(serializedApps)) {
            apps = new Gson().fromJson(serializedApps,appInfoTyp
e);
        }
        for (AppInfo app : apps) {
            subscriber.onNext(app);
        }
        subscriber.onCompleted();
    });
}
```

`getApps()` 方法返回一个 `AppInfo` 的Observable。它先从Android的SharePreferences读取到已安装的应用程序列表。反序列化，并一个接一个的发射AppInfo数据。使用新的方法来检索列表， `loadList()` 函数改成下面这样：

```
private void loadList() {
    mRecyclerView.setVisibility(View.VISIBLE);
    getApps().subscribe(new Observer<AppInfo>() {
        @Override
        public void onCompleted() {
            mSwipeRefreshLayout.setRefreshing(false);
            Toast.makeText(getActivity(), "Here is the list!", T
oast.LENGTH_LONG).show();
        }

        @Override
        public void onError(Throwable e) {
            Toast.makeText(getActivity(), "Something went wrong!"
, Toast.LENGTH_SHORT).show();
            mSwipeRefreshLayout.setRefreshing(false);
        }

        @Override
        public void onNext(AppInfo appInfo) {
            mAddedApps.add(appInfo);
                mAdapter.addApplication(mAddedApps.size() - 1, a
ppInfo);
        }
    });
}
```

如果我们运行代码， `StrictMode` 将会报告一个不合规操作，这是因为 `SharePreferences` 会减慢I/O操作。我们所需要做的是指定 `getApps()` 需要在调度器上执行：

```
getApps().subscribeOn(Schedulers.io())
        .subscribe(new Observer<AppInfo>() { [...]
```

`Schedulers.io()` 将会去掉 `StrictMode` 的不合规操作，但是我们的App现在崩溃了是因为：

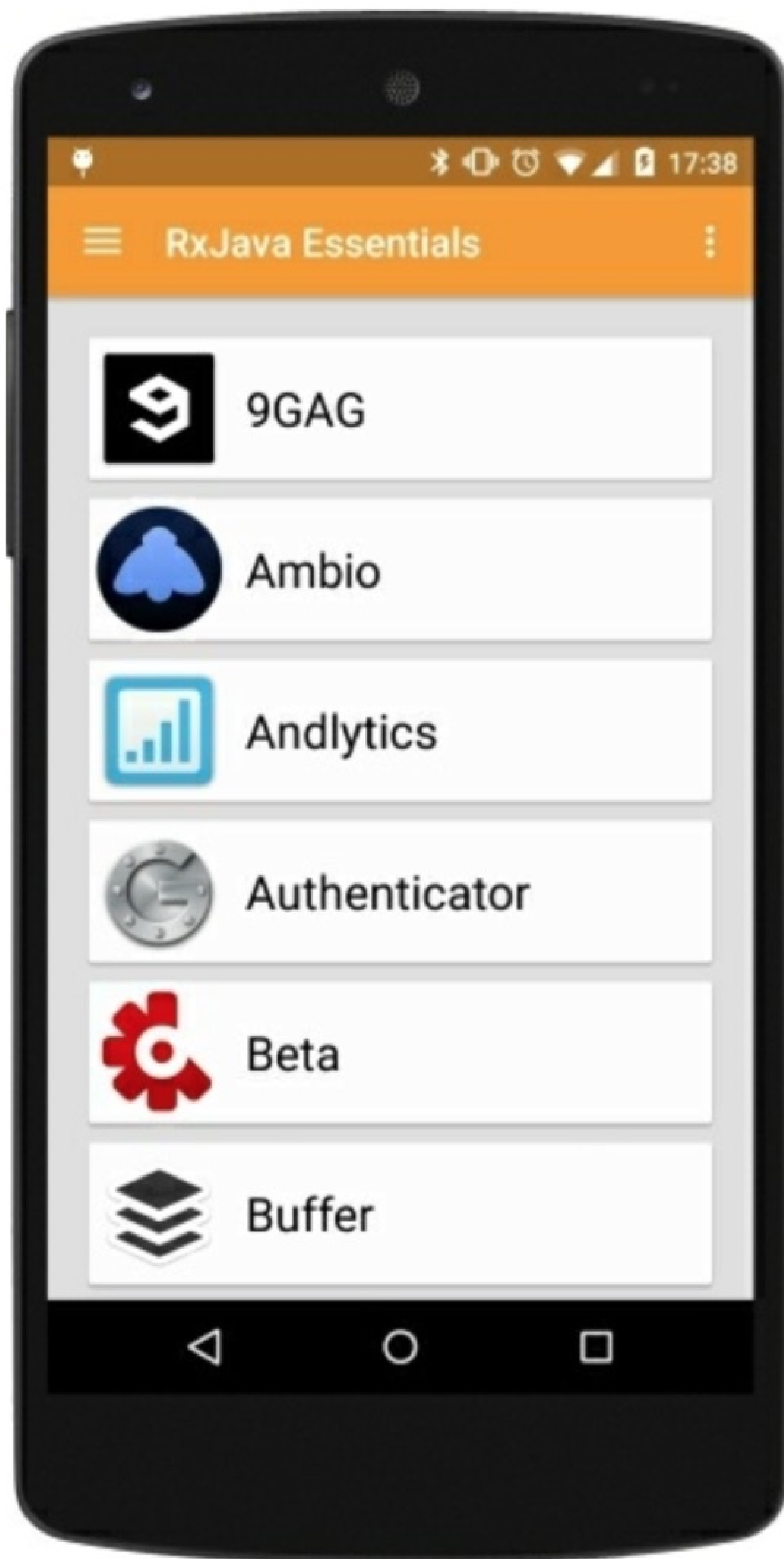
```
at rx.internal.schedulers.ScheduledAction.run(ScheduledAction.ja
v a:58)
at java.util.concurrent.Executors$RunnableAdapter.call(Executors
. java:422)
at java.util.concurrent.FutureTask.run(FutureTask.java:237)
at java.util.concurrent.ScheduledThreadPoolExecutor$ScheduledFut
u reTask.access$201(ScheduledThreadPoolExecutor.java:152)
at java.util.concurrent.ScheduledThreadPoolExecutor$ScheduledFut
u reTask.run(ScheduledThreadPoolExecutor.java:265)
at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolE
x ecutor.java:1112)
at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPool
E xecutor.java:587)
at java.lang.Thread.run(Thread.java:841) Caused by:
    android.view.ViewRootImpl$CalledFromWrongThreadException: On
ly the original thread that created a view hierarchy can touch i
ts views.
```

Only the original thread that created a view hierarchy can touch its views.

我们再次回到Android的世界。这条信息简单的告诉我们我们试图在一个非UI线程来修改UI操作。意思是我们需要在I/O调度器上执行我们的代码。因此我们需要和I/O调度器一起执行代码，但是当结果返回时我们需要在UI线程上操作。RxJava让你能够订阅一个指定的调度器并观察它。我们只需在 `loadList()` 函数添加几行代码，那么每一项就都准备好了：

```
getApps()
.onBackpressureBuffer()
.subscribeOn(Schedulers.io())
.observeOn(AndroidSchedulers.mainThread())
.subscribe(new Observer<AppInfo>() { [...]
```

`observeOn()` 方法将会在指定的调度器上返回结果：如例子中的UI线程。`onBackpressureBuffer()` 方法将告诉Observable发射的数据如果比观察者消费的数据要更快的话，它必须把它们存储在缓存中并提供一个合适的时间给它们。做完这些工作之后，如果我们运行App，就会出现已安装的程序列表：


17:38
RxJava Essentials
9GAG
Ambio
Andlytics
Authenticator
Beta
Buffer

处理耗时的任务

我们已经知道如何处理缓慢的I/O操作。让我们看一个与I/O无关的耗时的任务。例如，我们修改 `loadList()` 函数并创建一个新的 `slow` 函数发射我们已安装的app数据。

```
private Observable<AppInfo> getObservableApps(List<AppInfo> apps)
 {
    return Observable .create(subscriber -> {
        for (double i = 0; i < 1000000000; i++) {
            double y = i * i;
        }
        for (AppInfo app : apps) {
            subscriber.onNext(app);
        }
        subscriber.onCompleted();
    });
}
```

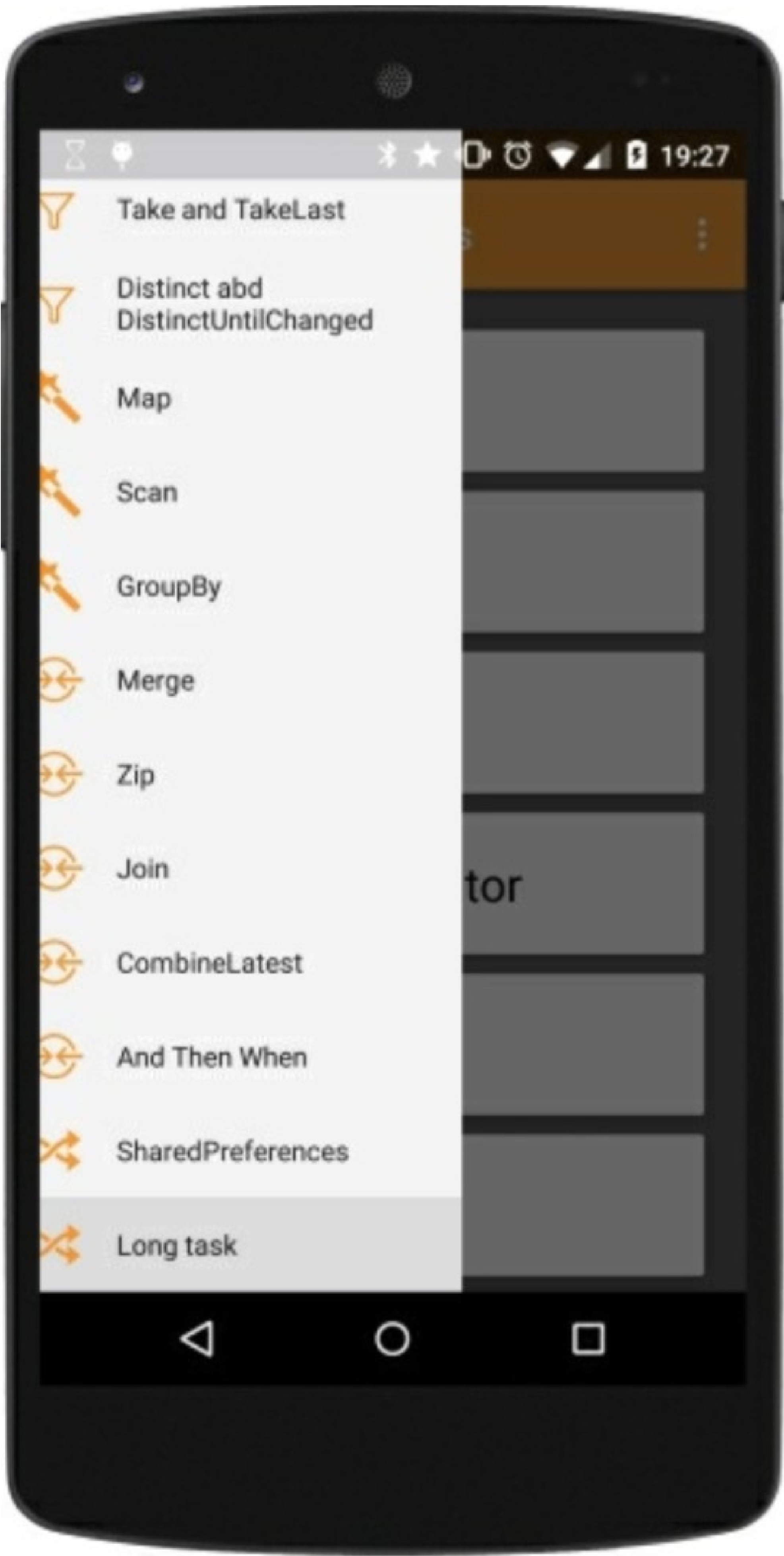
正如你看到的，这个函数执行了一些毫无意义的计算，只是针对这个例子消耗时间，然后从 `List<AppInfo>` 对象中发射我们的 `AppInfo` 数据，现在，我们重排 `loadList()` 函数如下：

```
private void loadList(List<AppInfo> apps) {
    mRecyclerView.setVisibility(View.VISIBLE);
    getObservableApps(apps)
        .subscribe(new Observer<AppInfo>() {
            @Override
            public void onCompleted() {
                mSwipeRefreshLayout.setRefreshing(false);
                Toast.makeText(getActivity(), "Here is the list!"
, Toast.LENGTH_LONG).show();
            }

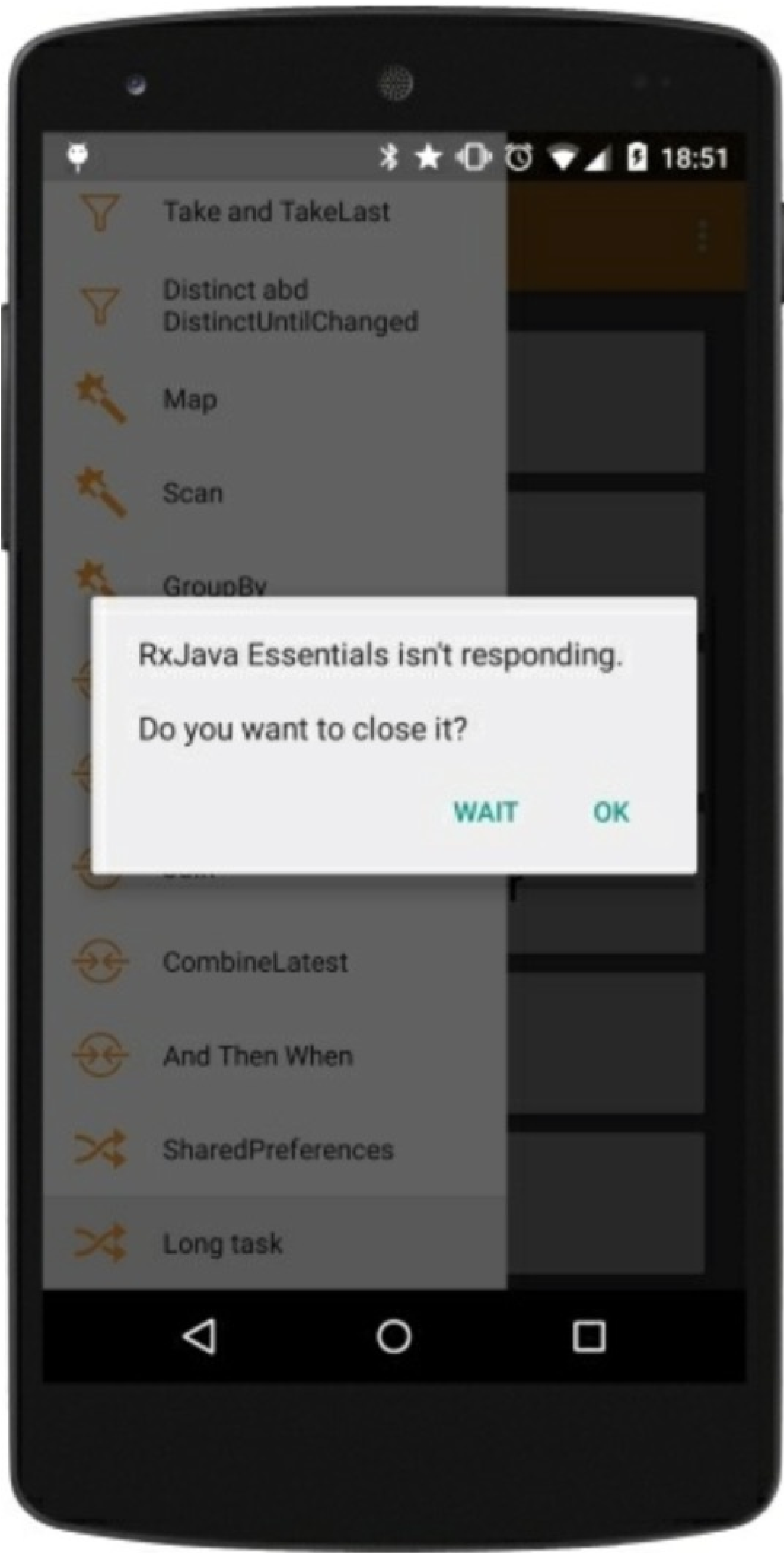
            @Override
            public void onError(Throwable e) {
                Toast.makeText(getActivity(), "Something went wr
ong!", Toast.LENGTH_SHORT).show();
                mSwipeRefreshLayout.setRefreshing(false);
            }

            @Override
            public void onNext(AppInfo appInfo) {
                mAddedApps.add(appInfo);
                mAdapter.addApplication(mAddedApps.size() - 1, a
ppInfo);
            }
        });
}
```

如果我们运行这段代码，当我们点击 `Navigation Drawer` 菜单项时App将会卡住一会，然后你能看到下图中半关闭的菜单:


19:27
Take and TakeLast
Distinct abd DistinctUntilChanged
Map
Scan
GroupBy
Merge
Zip
Join
CombineLatest
And Then When
SharedPreferences
Long task
tor

如果我们不够走运的话，我们可以看到下图中经典的ANR信息框：


18:51
Take and TakeLast
Distinct abd DistinctUntilChanged
Map
Scan
GroupBy
RxJava Essentials isn't responding.
Do you want to close it?
WAIT
OK
CombineLatest
And Then When
SharedPreferences
Long task

可以确定的是，我们将会看到下面在logcat中不愉快的信息：

```
I/Choreographer  Skipped 598 frames! The application may be doin
g too much work on its main thread.
```

这条信息比较清楚，Android在告诉我们用户体验非常差的原因是我们用不必要的工作量阻塞了UI线程。但是我们已经知道了如何处理它：我们有调度器！我们只须添加几行代码到我们的Observable链中就能去掉加载慢和 `Choreographer` 信息：

```
getObservableApps(apps)
    .onBackpressureBuffer()
    .subscribeOn(Schedulers.computation())
    .observeOn(AndroidSchedulers.mainThread())
    .subscribe(new Observer<AppInfo>() { [...]
```

用这几行代码，我们将可以快速关掉 `Navigation Drawer` ,一个漂亮的进度条，一个工作在独立的线程缓慢执行的计算任务，并在主线程返回结果让我们更新已安装的应用列表。

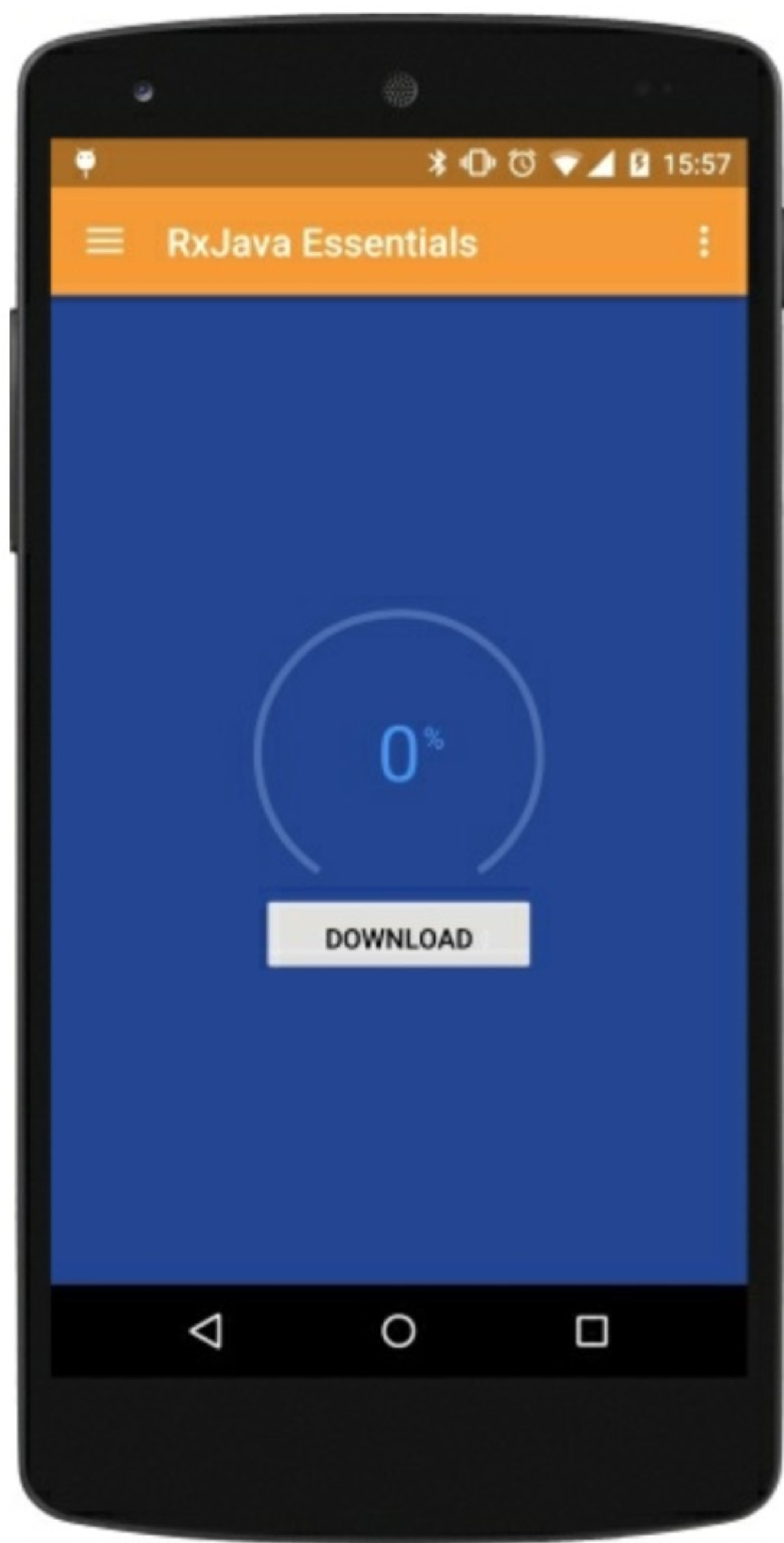
执行网络任务

在当今99%的移动应用中网络都是必不可缺的一部分：总是需要连接远程服务器来检索App需要的信息。

作为网络访问的第一个案例，我们将创建下面这样一个场景:

- 加载一个进度条。
- 用一个按钮开始文件下载。
- 下载过程中更新进度条。
- 下载完后开始视频播放。

我们的用户界面非常简单，我们只需要一个有趣的进度条和一个下载按钮。


15:57
RxJava Essentials
0%
DOWNLOAD

首先，我们创建 `mDownloadProgress`

```
private PublishSubject<Integer> mDownloadProgress = PublishSubje
ct.create();
```

这个主题我们用来管理进度的更新，它和 `download` 函数协同工作。

```
private boolean downloadFile(String source, String destination)
{
    boolean result = false;
    InputStream input = null;
    OutputStream output = null;
    HttpURLConnection connection = null;
    try {
        URL url = new URL(source);
        connection = (HttpURLConnection) url.openConnection();
        connection.connect();
        if (connection.getResponseCode() != HttpURLConnection.HT
TP_OK) {
            return false;
        }
        int fileLength = connection.getContentLength();
        input = connection.getInputStream();
        output = new FileOutputStream(destination);
        byte data[] = new byte[4096];
        long total = 0;
        int count;
        while ((count = input.read(data)) != -1) {
            total += count;
            if (fileLength >0) {
                int percentage = (int) (total * 100 / fileLength
);
                mDownloadProgress.onNext(percentage);
            }
            output.write(data, 0, count);
        }
        mDownloadProgress.onCompleted();
        result = true;
```

```
        } catch (Exception e) {
            mDownloadProgress.onError(e);
        } finally {
            try {
                if (output != null) {
                    output.close();
                }
                if (input != null) {
                    input.close();
                }
            } catch (IOException e) {
                mDownloadProgress.onError(e);
            }
            if (connection != null) {
                connection.disconnect();
                mDownloadProgress.onCompleted();
            }
        }
        return result;
    }
```

上面的这段代码将会触发 `NetworkOnMainThreadException` 异常。我们可以创建RxJava版本的函数进入我们挚爱的响应式世界来解决这个问题：

```
private Observable<Boolean> obserbableDownload(String source, St
ring destination) {
    return Observable.create(subscriber -> {
        try {
            boolean result = downloadFile(source, destination);
            if (result) {
                subscriber.onNext(true);
                subscriber.onCompleted();
            } else {
                subscriber.onError(new Throwable("Download faile
d."));
            }
        } catch (Exception e) {
            subscriber.onError(e);
        }
    });
}
```

现在我们需要触发下载操作，点击下载按钮:

```
@OnClick(R.id.button_download)
void download() {
    mButton.setText(getString(R.string.downloading));
    mButton.setClickable(false);
    mDownloadProgress.distinct()
    .observeOn(AndroidSchedulers.mainThread())
    .subscribe(new Observer<Integer>() {

        @Override
        public void onCompleted() {
            App.L.debug("Completed");
        }

        @Override
        public void onError(Throwable e) {
            App.L.error(e.toString());
        }

        @Override
```

```
        public void onNext(Integer progress) {
            mArcProgress.setProgress(progress);
        }
    });

    String destination = "sdcardsoftboy.avi";
    obserbableDownload("http://archive.blender.org/fileadmin/mov
ies/softboy.avi", destination)
        .subscribeOn(Schedulers.io())
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe(success -> {
            resetDownloadButton();
            Intent intent = new Intent(android.content.Intent.AC
TION_VIEW);
            File file = new File(destination);
            intent.setDataAndType(Uri.fromFile(file),"video/avi"
);
            intent.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
            startActivity(intent);
        }, error -> {
            Toast.makeText(getActivity(), "Something went south"
, Toast.LENGTH_SHORT).show();
            resetDownloadButton();
        });
}
```

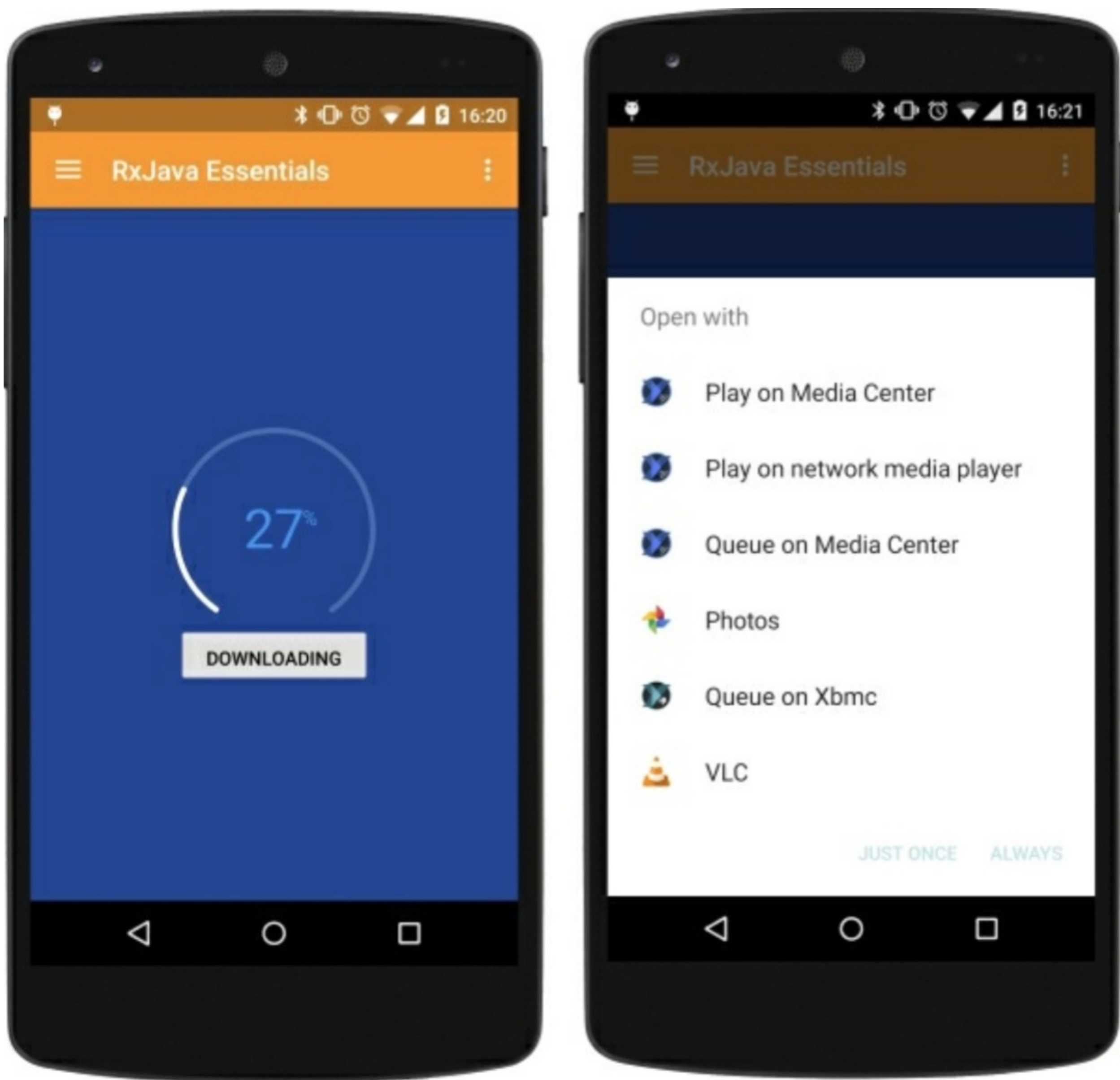
我们使用Butter Knife的注解 `@OnClick` 来绑定按钮的方法并更新按钮信息和点击状态：我们不想让用户点击多次从而触发多次下载事件。

然后，我们创建一个subscription来观察下载进度并相应的更新进度条。很明显，我们订阅在主线程是因为进度条是UI元素。

```
obserbableDownload("http://archive.blender.org/fileadmin/movies/
softboy.avi", "sdcardsoftboy.avi";)
```

这是一个下载Observable。网络调用是一个I/O任务，理应使用I/O调度器。当下载完成，就会在 `onNext()` 中启动视频播放器，并且播放器将会在目标路径找到下载的文件.。

下图展示了下载进度和视频播放器选择对话框：



总结

这一章中，我们学习了如何简单的将多线程应用在我们的App中。RxJava为此提供了极其实用的工具：调度器。调度器以及不同应用场景下的优化方案一起，将我们从 `StrictMode` 中的不合法操作以及阻塞I/O的方法中解放出来。我们现在可以用简单的，响应式的，并在整个App中保持一致的方式来访问本地存储和网络。

下一章中，我们将会冒更大的险来创建一个正儿八经的App，并使用Square公司开源的REST API库Retrofit来获取不同的远程数据并创建一个复杂的material design UI。

与REST无缝结合-RxJava和Retrofit

在上一章中，我们学习了如何使用调度器在不同于UI线程的线程上操作。我们学习了如何高效的运行I/O任务而不用阻塞UI以及如何运行耗时的计算任务而不耗损应用性能。在最后一章中，我们将创建一个最终版的应用实例，用Retrofit映射远程API,异步查询数据，轻松创造一个丰富的UI。

项目目标

我们将在已有的例子中创建一个新的 `Activity` 。这个 `Activity` 将通过StackExchange API从stackoverflow检索出最活跃的10位用户。App使用这些信息来展示一个包含用户头像、姓名、名望数以及住址的列表。对每一位用户，app使用OpenWeatherMap API来检索该用户住址当地的天气预报，并显示一个小天气图标。基于从StackOverflow检索的信息，app对列表中的每一位用户提供一个 `onClick` 事件，打开他们在个人信息中设定的个人网站或者Stack Overflow的个人主页。

Retrofit

Retrofit是Square公司专为Android和Java设计的一个类型安全的REST客户端。它帮助你轻松地与任意REST API交互，并完美兼容RxJava：所有的JSON响应对象都被映射成原始的Java对象，并且所有的网络调用都基于Rxjava Observable这些对象。

使用API文档，我们可以定义我们从服务器接收的JSON响应数据。为了很容易的将JSON响应数据映射为我们的Java代码，我们将使用jsonschema2pojo, 这个服务将灵活地生成所有与JSON响应数据相映射的Java类。

当我们把所有的Java model准备好后，我们就可以开始建立Retrofit。Retrofi使用标准的Java接口来映射API路由。例如例子中，我们将使用来自API的一个路由，下面是我们Retrofit的接口:

```
public interface StackExchangeService {
    @GET("/2.2/users?order=desc&sort=reputation&site=stackoverflow")
    Observable<User sResponse> getMostPopularSOusers(@Query("pagesize") int howmany);
}
```

`interface` 接口只包含一个方法，即 `getMostPopularSOusers` 。这个方法用整型 `howmany` 作为一个参数并返回 `UserResponse` 的Observable。

当我们有了 `interface` ，我们可以创建 `RestAdapter` 类，为了更清楚的组织我们的代码，我们创建一个 `SeApiManager` 函数提供一种更适当的方式来和StackExchange API交互。

```
public class SeApiManager {
    private final StackExchangeService mStackExchangeService;

    public SeApiManager() {
        RestAdapter restAdapter = new RestAdapter.Builder()
            .setEndpoint("https://api.stackexchange.com")
            .setLogLevel(RestAdapter.LogLevel.BASIC)
            .build();
        mStackExchangeService = restAdapter.create(StackExchange
Service.class);
    }

    public Observable<List<User>> getMostPopularSOusers(int howm
any) {
        return mStackExchangeService
        .getMostPopularSOusers(howmany)
        .map(UsersResponse::getUsers)
        .subscribeOn(Schedulers.io())
        .observeOn(AndroidSchedulers.mainThread());
    }
}
```

为了简化例子，我们不再将这个类设计为它本该设计成的单例。使用依赖注入解决方案，如Dagger2，可使代码质量更高。

创建 `RestAdapter` 类，需要对客户端API设置几个重要的方面。这个例子中，我们设置了 `endpoint` 和 `log level` 。由于这个例子中URL只是硬编码，像这样使用外部资源来存储数据很重要。避免在代码中硬编码字符串是一个好的实践。

Retrofit把 `RestAdapter` 类和我们的API接口绑定在一起后就完成了创建。它返回给我们一个对象用来请求API。我们可以选择直接暴露这个对象，或者以某种封装方式来限制对它的访问。在这个例子中，我们封装它并只暴露 `getMostPopularSOusers` 方法。这个方法执行查询，使用Retrofit解析JSON响应数据。获得用户列表，并返回给订阅者。如你所见，使用Retrofit、RxJava和Retrolambda，我们几乎没有模板代码：它非常紧凑而且可读性很高。

现在，我们已经有一个API管理者来提供一个响应式的方法，它从远程API获取数据并给I/O调度器，解析映射最后为我们的消费者提供一个简洁的用户列表。

```
```

App架构

我们不使用任何MVC，MVP，或者MVVM模式。因为那不是这本书的目的，因此我们的 `Activity` 类将包含我们需要创建和展示用户列表的所有逻辑。

创建Activity类

我们将在 `onCreate()` 方法里创建 `SwipeRefreshLayout` 和 `RecyclerView` ；我们有一个 `refreshList()` 方法来处理用户列表的获取和展示， `showRefreshing()` 方法来管理进度条和 `RecyclerView` 的显示。

我们的 `refreshList()` 函数看起来如下：

```
private void refreshList() {
    showRefresh(true);
    mSeApiManager.getMostPopularSOusers(10)
        .subscribe(users -> {
            showRefresh(false);
            mAdapter.updateUsers(users);
        }, error -> {
            App.L.error(error.toString());
            showRefresh(false);
        });
}
```

我们显示了进度条，从StackExchange API 管理器观测用户列表。一旦获取到列表数据，我们开始展示它并更新 `Adapter` 的内容并让 `RecyclerView` 显示为可见。

创建RecyclerView Adapter

我们从REST API获取到数据后，我们需要把它绑定View上，并用一个适配器填充列表。我们的RecyclerView适配器是标准的。它继承于 `RecyclerView.Adapter` 并指定它自己的 `ViewHolder` ：

```
public static class ViewHolder extends RecyclerView.ViewHolder {
    @InjectView(R.id.name) TextView name;
    @InjectView(R.id.city) TextView city;
    @InjectView(R.id.reputation) TextView reputation;
    @InjectView(R.id.user_image) ImageView user_image;
    public ViewHolder(View view) {
        super(view);
        ButterKnife.inject(this, view);
    }
}
```

我们一旦收到来自API管理器的数据，我们可以设置界面上所有的标签： `name` , `city` 和 `reputation` 。

为了展示用户的头像，我们将使用Sergey Tarasevich写的Universal Image Loader。实践证明，UIL是非常有名的好用的图片管理库。我们也可以使用Square公司的Picasso，Glide或者Facebook公司的Fresco。取决于你自己的喜好。最关键的是无需重复造轮子：库能够方便开发者并让他们更快速实现目标。

在我们的适配器中，我们可以这样：

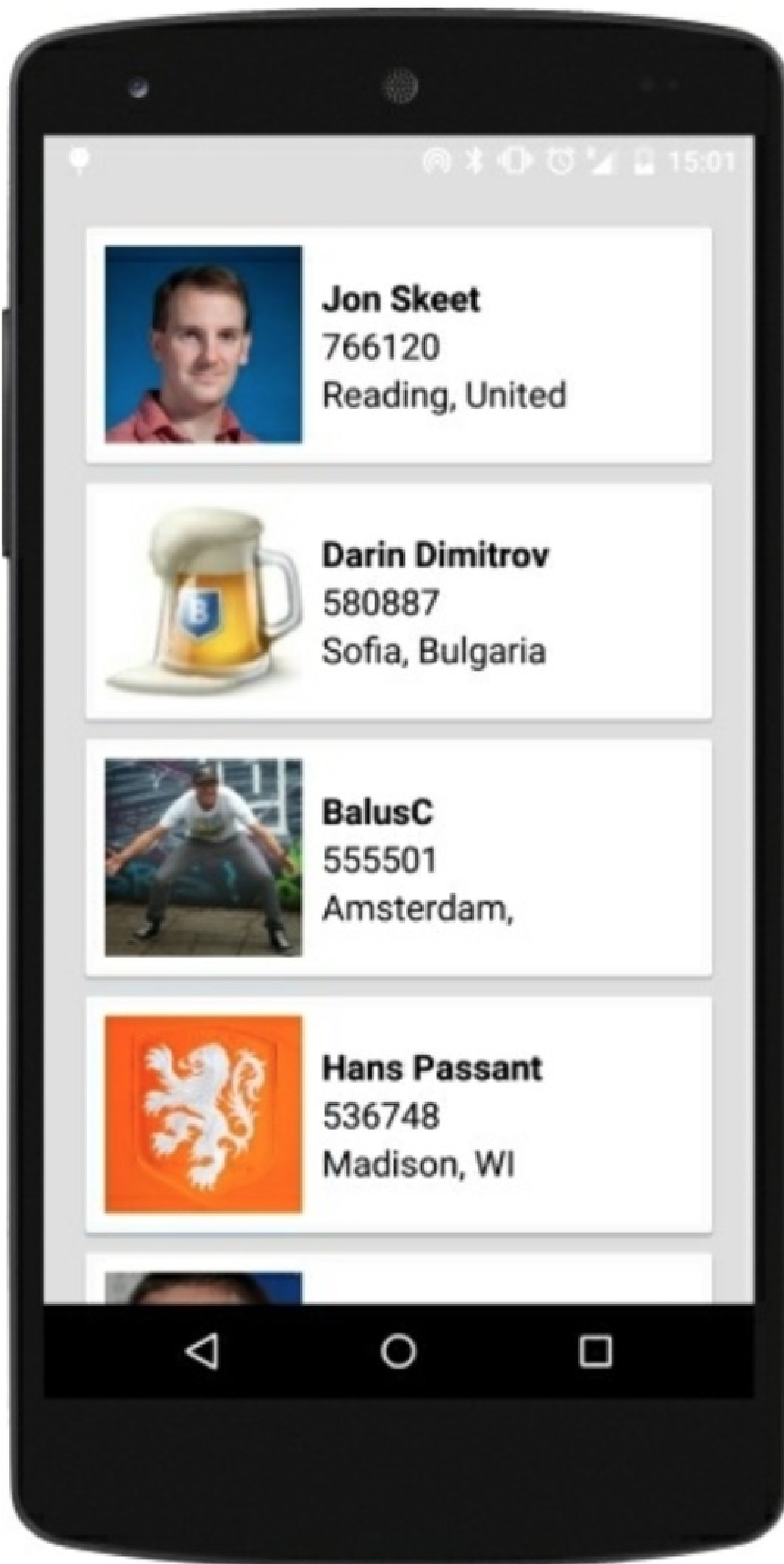
```
@Override
public void onBindViewHolder(SoAdapter.ViewHolder holder, int po
sition) {
    User user = mUsers.get(position);
    holder.setUser(user);
}
```

在 `ViewHolder` ，我们可以这样：

```
public void setUser(User user) {
    name.setText(user.getDisplayName());
    city.setText(user.getLocation());
    reputation.setText(String.valueOf(user.getReputation()));

    ImageLoader.getInstance().displayImage(user.getProfileImage(
), user_image);
}
```

此时，我们可以允许代码获得一个用户列表，正如下图所示：


15:01
Jon Skeet
766120
Reading, United
Darin Dimitrov
580887
Sofia, Bulgaria
BalusC
555501
Amsterdam,
Hans Passant
536748
Madison, WI

检索天气预报

我们加大难度，将当地城市的天气加入列表中。**OpenWeatherMap**是一个灵活公共在线天气API，我们可以查询许多有用的预报信息。

和往常一样，我们将使用Retrofit映射到API然后通过RxJava来访问它。至于StackExchange API，我们将创建 `interface` ， `RestAdapter` 和一个灵活的管理器：

```
public interface OpenWeatherMapService {
    @GET("data2.5/weather")
    Observable<WeatherResponse> getForecastByCity(@Query("q") String city);
}
```

这个方法用城市名字作为参数提供当地的预报信息。我们像下面这样将接口和 `RestAdapter` 类绑定在一起：

```
RestAdapter restAdapter = new RestAdapter.Builder()
        .setEndpoint("http://api.openweathermap.org")
        .setLogLevel(RestAdapter.LogLevel.BASIC)
        .build();
mOpenWeatherMapService = restAdapter.create(OpenWeatherMapService.class);
```

像以前一样，我们只有两件事需要立马去做：设置API端口和log级别。

`OpenWeatherMapApiManager` 类将提供下面的方法：

```
public Observable<WeatherResponse> getForecastByCity(String city) {
    return mOpenWeatherMapService.getForecastByCity(city)
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread());
}
```

现在，我们有了用户列表，我们可以根据城市名来查询OpenWeatherMap获得天气预报信息。下一步是修改我们的 `ViewHolder` 类来为每位用户展示相应的天气图标。

我们使用这些工具方法先验证用户主页信息并获得一个合法的城市名字：

```
private boolean isCityValid(String location) {
    int separatorPosition = getSeparatorPosition(location);
    return !"".equals(location) && separatorPosition > -1;
}

private int getSeparatorPosition(String location) {
    int separatorPosition = -1;
    if (location != null) {
        separatorPosition = location.indexOf(",");
    }
    return separatorPosition;
}

private String getCity(String location, int position) {
    if (location != null) {
        return location.substring(0, position);
    } else {
        return "";
    }
}
```

借助一个有效的城市名，我们可以用下面命令来获得我们所需要天气的所有数据：

```
OpenWeatherMapApiManager.getInstance().getForecastByCity(city)
```

用天气响应的结果，我们可以获得天气图标的URL：

```
getWeatherIconUrl(weatherResponse);
```

用图标URL，我们可以检索到图标本身：

```
private Observable<Bitmap> loadBitmap(String url) {
    return Observable.create(subscriber -> {
        ImageLoader.getInstance().displayImage(url,city_image, n
ew ImageLoadingListener() {
            @Override
            public void onLoadingStarted(String imageUri, View v
iew) {
            }

            @Override
            public void onLoadingFailed(String imageUri, View vi
ew, FailReason failReason) {
                subscriber.onError(failReason.getCause());
            }

            @Override
            public void onLoadingComplete(String imageUri, View
view, Bitmap loadedImage) {
                subscriber.onNext(loadedImage);
                subscriber.onCompleted();
            }

            @Override
            public void onLoadingCancelled(String imageUri, View
 view) {
                subscriber.onError(new Throwable("Image loading
cancelled"));
            }
        });
    });
}
```

这个 `loadBitmap()` 返回的Observable可以链接前面一个，并且最后我们可以为这个任务返回一个单独的Observable：

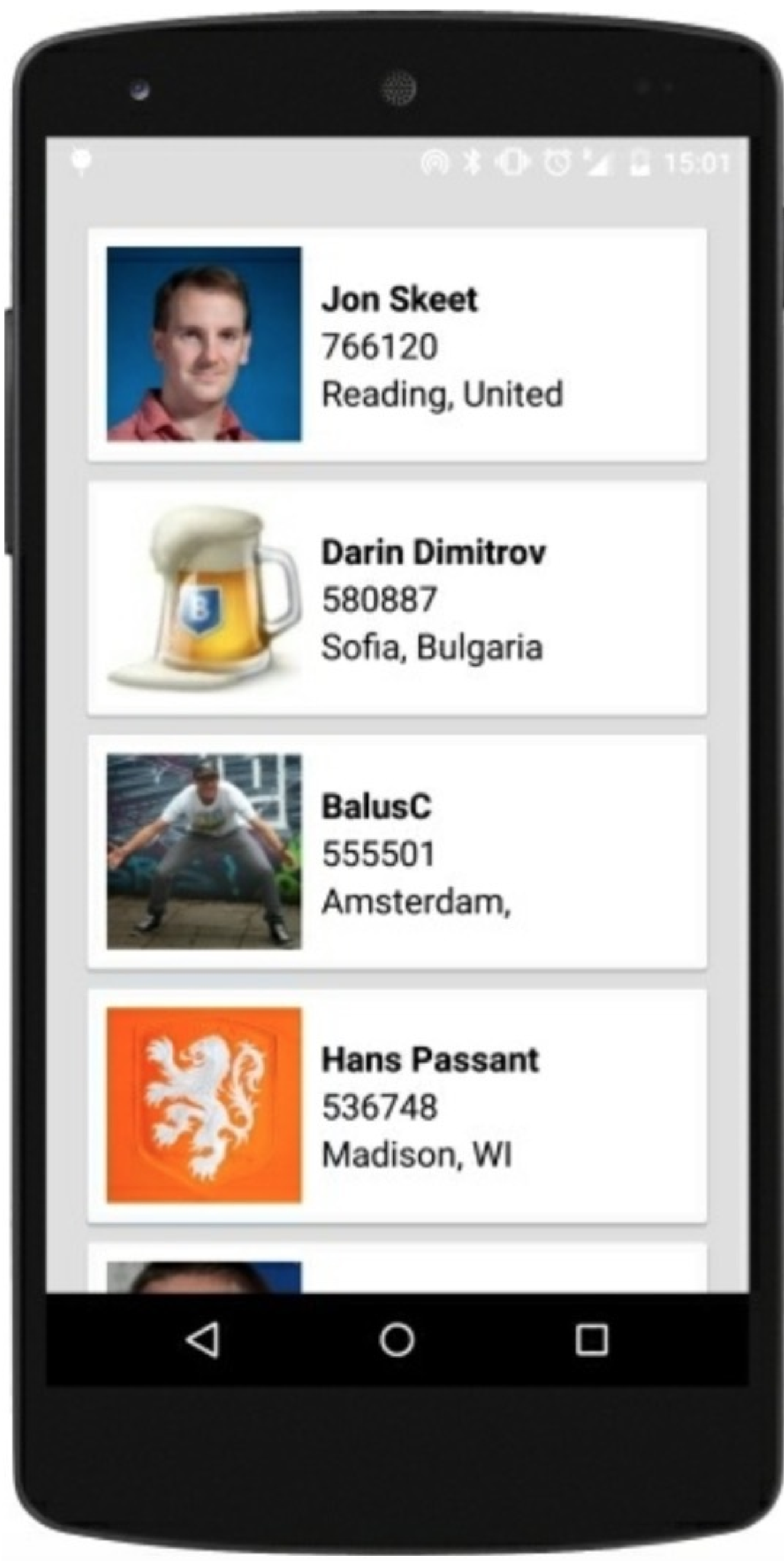
```
if (isCityValid(location)) {
    String city = getCity(location, separatorPosition);
    OpenWeatherMapApiManager.getInstance().getForecastByCity(cit
y)
        .filter(response -> response != null)
        .filter(response -> response.getWeather().size() > 0)
        .flatMap(response -> {
            String url = getWeatherIconUrl(response);
            return loadBitmap(url);
        })
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
    .subscribe(new Observer<Bitmap>() {

        @Override
        public void onCompleted() {
        }

        @Override
        public void onError(Throwable e) {
            App.L.error(e.toString());
        }

        @Override
        public void onNext(Bitmap icon) {
            city_image.setImageBitmap(icon);
        }
    });
}
```

运行代码，我们可以在下面列表中为每个用户获得新的天气图标：


15:01
Jon Skeet
766120
Reading, United
Darin Dimitrov
580887
Sofia, Bulgaria
BalusC
555501
Amsterdam,
Hans Passant
536748
Madison, WI

打开网站

使用用户主页包含的信息，我们将会创建一个 `onClick` 监听器来导航到用户web页面，如果有的话，否则打开在Stack Overflow上的个人主页。

为了实现它，我们简单实现 `Activity` 类的接口，用来在适配器触发Android的 `onClick` 事件。

我们的 `Adapter ViewHolder` 指定这个接口：

```
public interface OpenProfileListener {
    public void open(String url);
}
```

`Activity` 实现它：

```
[...] implements SoAdapter.ViewHolder.OpenProfileListener { [...
]
    mAdapter.setOpenProfileListener(this);
[...]

@Override
public void open(String url) {
    Intent i = new Intent(Intent.ACTION_VIEW);
    i.setData(Uri.parse(url));
    startActivity(i);
}
```

`Activity` 收到URL并用外部Android浏览器打开它。我们的 `ViewHolder` 负责在用户列表的每个卡片上创建 `OnClickListener` 并检查我们是打开Stack Overflow用户主页还是外部个人站：

```
mView.setOnClickListener(view -> {
    if (mProfileListener != null) {
        String url = user.getWebsiteUrl();
        if (url != null && !url.equals("") && !url.contains("search")) {
            mProfileListener.open(url);
        } else {
            mProfileListener.open(user.getLink());
        }
    }
)}；
```

一旦我们点击了，我们将直接重定向到预期的网站。在Android上，我们可以用RxAndroid的一种特殊形式（ViewObservable）以更加响应式的方式实现同样的结果。

```
ViewObservable.clicks(mView)
    .subscribe(onClickEvent -> {
    if (mProfileListener != null) {
        String url = user.getWebsiteUrl();
        if (url != null && !url.equals("") && !url.contains("search")) {
            mProfileListener.open(url);
        } else {
            mProfileListener.open(user.getLink());
        }
    }
});
```

上面两块代码片段是等价的，你可以选择最喜欢的方式来实现。

总结

我们的旅程结束了。相信你已经准备好将你的Java应用带到一个新的代码质量水平。你可以享受一个新的编程模式并把更流畅的思维方式应用到日常编程生活中。RxJava提供了一种以面向时序的方式考虑数据的机会：所有事情都是持续变化的，数据在更新，事件在触发，然后你就可以创建事件响应式的、灵活的、运行流畅的App。

刚开始切换到RxJava看起来困难并且耗时，但我们已经体验到了如何通过响应式的方式有效地处理日常问题。现在你可以把你的旧代码迁移到RxJava上:给这些同步 `getters` 一种新的响应式。

RxJava是一个正在不断发展和扩大的世界。还有许多方法我们还没有去探索。有些方法甚至还没有，通过RxJava，你可以创建你自己的操作符并把他们发展地更远。

Android是一个好玩的地方，但是它也有局限性。作为一个Android开发者，你可以用RxJava和RxAndroid克服其中的许多。我们用AndroidScheduler只简单提了下RxAndroid,除了在最后一章，你了解了 `ViewObservable` 。RxAndroid给了你许多：例如， `WidgetObservable` ， `LifecycleObservable` 。往后将它发展地更长远的任务就取决于你了。

谨记可观测序列就像一条河：它们是流动的。你可以“过滤”(filter)一条河，你可以“转换”(transform)一条河，你可以将两条河合并(combine)成一个，然后依然畅流如初。最后，它就成了你想要的那条河。

```
“Be Water，my friend”

            --Bruce Lee
```